Proc. Estonian Acad. Sci. Engin., 1995, 1, 2, 139–157
https://doi.org/10.3176/eng.1995.2.03

ATTRIBUTED MODELS OF COMPUTING

Merik MERISTE[a] and Jaan PENJAM[b]

[a] Tartu Ülikooli arvutiteaduse instituut (Computer Science Department, University of Tartu), Liivi 2, EE-2400 Tartu, Eesti (Estonia)
[b] Küberneetika Instituut (Institute of Cybernetics), Akadeemia tee 21, EE-0026 Tallinn, Eesti (Estonia)

Presented by J. Engelbrecht

Received 17 October 1995, revised 23 January 1996, accepted 31 January 1996

Abstract. This paper introduces the attributed automata (AA) as a formalism for executable specification of knowledge, using regular syntax with attributes to represent contextual relations and the semantic properties of concepts. AA are treated as a generalisation of a state transition network, with attributes and computational relations attached to states and transitions, respectively. The attributed automaton model provides additional tools for the restructuring of large systems to reduce their conceptual complexity for simpler implementations. The paper discusses the general properties, composition/decomposition, and minimisation of AA.

Key words: attributed automata, executable specification, language recognizers, compositions of automata.

1. INTRODUCTION

Regular and context-free structures are classical and can be efficiently implemented as models of data structures. Automated computing is often successful to an extent adequately regular and/or context-free (surface or deep) substructures are extracted from the remaining structure of the data. Formal models integrating these structures with others and supporting restructuring of data, analysis and implementation are methodologically important. The modifications of models known in the theory of formal languages and automata (e.g., finite and pushdown automata and state transition systems) and the corresponding declarative formalisms (e.g.,

[a] Partially supported by the Grant no 1718 from the Estonian Science Foundation
[b] Partially supported by the Grant no 475 from the Estonian Science Foundation

attribute grammars and graph grammars) are widely used to achieve effectively executable specifications.

This paper discusses attributed automata (AA) as an extension to state transition systems. Different modifications of finite automata and state transition systems use additional memory and computation rules to handle contextual and semantic information [1–3]. All such extensions are covered in a problem-oriented manner, where the basic formalism and its properties are treated as a secondary matter. This paper studies the attachment of memory to a state transition network. The paper focuses on how the adding of attributes will affect the expressive power and composition/decomposition of networks.

The attributed automaton model provides additional tools for restructuring large state transition systems, allowing one to reduce the number of states in a transition system and the conceptual complexity of their specification. Due to their properties, AA have interesting applications in pattern recognition, language processing and in the specification of distributed systems. Here the composition theory of AA and different specific models supporting AA applications provide a basic framework for technological environments.

The next section introduces the concept of an attributed automaton and discusses some special cases and the expressive power. It is followed by a motivation for composition/decomposition of AA and an overview of morphisms on AA and problems of behavioural equivalence of AA. In the final section, some related engineering problems are described.

2. ATTRIBUTED AUTOMATA

2.1. Definition

An attributed automaton is a state transition system with attributes and computational relations attached to states and transitions, respectively. Generally speaking, domains of attributes are free, and they can either be of a primitive type or a higher type with a complex structure. Eventually, this allows us to ignore the input tape of the classical finite automaton. The input can be modelled by distinguishing a special (input) attribute and the transformation functions. This will be discussed later in greater detail.

Definition 2.1. *An attributed automaton is a transition network* $M = (S, T)$, *where:*

- S *is a set of states with two distinguished subsets:* $S_0 \subseteq S$, *the initial states, and* $S_f \subseteq S$, *the final states*;
  every state $s \in S$ *is associated with an attribute* a_s *that is a variable over the domain* A_s;

- $T \subseteq S \times S$ *is a set of transitions* ;
  every transition $t = (s, s') \in T$ *is associated with*

– *an enabling predicate* $P_t : A_s \longrightarrow$ **bool** *and*

– *a transformation function* $f_t : A_s \longrightarrow A_{s'}$.

Both the enabling predicates and transformation functions must be computable, i.e., partially recursive functions.

In an initial state, with an initial attribute value $x \in A_{s_0}$, the functioning of the automaton M is considered as a successive change of the current state. Transition from one state to another is possible only if the corresponding enabling predicate is true. Every transition includes an evaluation of the attribute of the next state, using the associated transformation function.

Similar to the classical automata, one can define a pair (s, a), where $s \in S$ and $a \in A_s$, as a *configuration* of the automaton M. Let us denote the set of all possible configurations of the automaton M by C_M. We suppress the subscript M if confusion is unlikely. In particular, (s_0, x) and (s_f, y) with $s_0 \in S_0$ and $s_f \in S_f$ are called *initial* and *final configurations*, respectively.

Now, the automaton M can be considered as a formal computing device with its operating cycle running over multiple configurations.

Definition 2.2. *The move from a configuration to the next one is a binary relation* $\mapsto$ *on the set* C *such that* $c \mapsto c'$ *iff there exists a transition* $t = (s, s') \in T$ *that* $P_t(a)$ *is true and* $f_t(a) = b$ *for* $c = (s, a)$ *and* $c' = (s', b)$.

To illustrate it, we can represent AA as transition graphs, where nodes correspond to the states and arcs to the transitions. The states are labelled by the associated attribute and the arcs by the enabling predicate in brackets (the predicate, which is equally **true**, can be omitted) and the transformation functions as in Fig. 1.

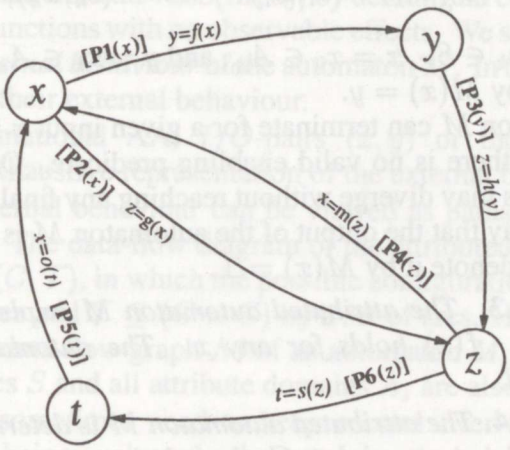


Fig. 1. Transition graph.

In the framework of the AA model, we can consider a traditional finite automaton [4] as a special case. In this kind of an attributed automaton,

all states have the same attribute domain (i.e., the finite input and output alphabets) with appropriate specific operations (see also section 2.3). Moreover, a finite automaton can be presented as an attributed automaton with one state [5]. On the other hand, an attributed automaton with finite domains of attributes can be presented as a finite automaton [6].

2.2. Behaviour of attributed automata

AA can be used for modelling of transformational as well as interactive systems. In the first case, AA behave like functions for computing one object from another. Here it is important to have a reachable final configuration because only the terminating transformations are desirable.

Without loss of generality, we can limit ourselves to AA with no transitions going out from the final states. Alternatively, we can introduce a new final state s' with its attribute domain $A_{s'} = A_s$, for the original final state s and outcoming transitions $t_1, t_2, \ldots, t_n$ with corresponding enabling predicates $P_{t_1}, P_{t_2}, \ldots, P_{t_n}$. The state s should be excluded from the set of final states. Introduction of a new transition $t = (s, s')$ with the enabling predicate $P_t(a_s) = \neg(P_{t_1}(a_s) \vee P_{t_2}(a_s) \vee \ldots \vee P_{t_n}(a_s))$ completes the construction. The automaton terminates its computations when it arrives at a final configuration $(s', y) \in S_F \times A_{s'}$. The attribute value y is called the *output* of this computation. Similarly, x is called the *input* if the computation started from an initial configuration (s_0, x).

The attributed automaton M transforms the input x into the output y if the reflexive transitive closure $\mapsto^*$ contains a pair (c_0, c_k) of an initial configuration $c_0 = (s_0, x)$ and a final configuration $c_k = (s_k, y)$. In other words, there should be a sequence

$$(s_0, x_0) \mapsto (s_1, x_1) \mapsto \ldots \mapsto (s_k, x_k),$$

where $s_0 \in S_0$, $s_k \in S_F$, $x = x_0 \in A_{s_0}$, and $y = x_k \in A_{s_k}$. We will denote this expression by $M(x) = y$.

The automaton M can terminate for a given input x in some non-final configuration if there is no valid enabling predicate. On the other hand, the computations may diverge without reaching any final configuration. In both cases, we say that the output of the automaton M is undefined for the input x, and we denote it by $M(x) = \perp$.

Definition 2.3. *The attributed automaton M implements a function f iff $M(x) = f(x)$ holds for any x. The automaton M is called transformational.*

Definition 2.4. *The attributed automaton M is deterministic if for any three configurations $c_1, c_2, c_3 \in C$, the relations $c_1 \mapsto c_2$ and $c_1 \mapsto c_3$ imply $c_2 \equiv c_3$. Otherwise, the automaton M is called non-deterministic.*

Non-deterministic transformational AA implement stochastic functions. Termination is not important for the so-called *interactive* systems. Even when they terminate and produce an object y from an

object x, it is much more important for the system to react to changes in its environment adequately. Generally, these changes cannot be predicted for a system which recognizes them by checking periodically (i.e., reading data from the environment) whether special events happened. The system responds by performing internal changes which, in turn, are registered by the environment as some internal event of the system. Frequently, internal events cause the generation of output data (or some "piece of result") to the environment.

Let us call the AA simulating interactive systems above *interactive AA*. In this case, the sequence of internal events (i.e., transformations of attribute instances) must synchronise with external events (i.e., events in the environment). This provides a trace of the computations to characterise the behaviour of the interactive AA. We define the sequence

$$Trace(M, c_0) = c_0 f_1 c_1 f_2 c_2 \ldots,$$

where for any $i > 0$, the moves $c_{i-1} \mapsto c_i$ can take place according to the given transitions $t_i = (s_{i-1}, s_i)$ for $c_0 = (s_0, x_0), c_1 = (s_1, x_1), \ldots \in C$, and c_0 is an initial configuration. We shall call the $Trace(M, c_0)$ a trace of the *internal behaviour* of the attributed automaton M. Traces may be infinite for interactive and diverging transformational AA.

Following the behaviour of AA with internal traces is practically impossible for larger systems. For an abstraction, internal events are usually classified as externally observable or non-observable. Observable events communicate with an environment, for example, the association of input and output of data. In our model, the communication between an automaton and the environment can be implemented as a side-effect of some transformation functions. Let us denote by $trace(M) = f_{i_1}, f_{i_2}, \ldots$ obtained from a $Trace(M, c_0)$ by deleting all configurations and transformation functions with no observable effects. We say that $trace(M)$ represents an *external behaviour* of the automaton M. In Section 4, we will compare AA by their external behaviour.

For transformational AA, I/O-pairs (x, y) or the functions they implement are exhaustive representation of the external behaviour.

Traces of internal behaviour can be viewed as paths in the data-flow diagrams of AA. The data-flow diagram of the attributed automaton M is the graph $\mathcal{M} = (C, \mathcal{T})$, in which the possible configurations C are formed by a set of vertices and $\mathcal{T} \subseteq (C \times C)$ by a set of arcs. Here, $(c_1, c_2) \in \mathcal{T}$ iff $c_1 \mapsto c_2$. The data-flow graph $\mathcal{M}$ of an automaton $M = (S, T)$ is finite iff the set of states S and all attribute domains A_s are also finite.

Further, we sometimes need to keep control over the states whose attributes are not being evaluated, when studying the behaviour of AA by a data-flow diagram. For that reason, the following projections of the relation $\mathcal{T}$ are used [1]:

[1] The functions $l(X)$ and $r(X)$ are sets of left and right components of a set of pairs X, e.g., $l(X) = \{z | (z, x) \in X\}$.

- *a transition relation*: $\tau \subseteq C \times l(C)$ and
- *an attribute relation*: $\alpha \subseteq C \times r(C)$.

Thus, $\tau(c) = s'$ and $\alpha(c) = b$ iff $c \mapsto c'$, where $c' = (s', b)$.

Next, we shall outline the problems related to the executable specifications in the AA framework:

- expressive power of AA;
- internal behaviour of AA, homomorphisms on AA;
- externally similar behaviour of AA, simulating attributed automaton;
- minimisation of specific types of AA.

2.3. Attributed automata as recognizers

A language recognizer is an application of an AA model. This provides a key to the indirect estimation of the expressive power of the formalism.

For language recognition applications, we need a specialisation of the attributed automaton, where attributes of states have one component I over domain Σ^* (strings in the input alphabet Σ) and operations $head : \Sigma^* \longrightarrow \Sigma$ and $tail : \Sigma^* \longrightarrow \Sigma^*$. These operations are commonly defined as:

$$head(\epsilon) = tail(\epsilon) = \epsilon, \quad head(aw) = a, \quad \text{and} \quad tail(aw) = w,$$

where $a \in \Sigma$, ϵ is an empty string, and both of these functions can be called from the enabling predicates as well as from the transformation functions. Let us suppose that a recognizer has only one initial and one final state. Any attribute of the initial state except the component I is set to a fixed value. The initial value of I is a string to be accepted. The specialisation can be defined as follows.

Definition 2.5. *A language recognizer is an attributed automaton* $M = (S, T, s_0, x, s_f)$, *where*

- S *is a set of states; every state* $s \in S$ *has an attribute with components* $a_1^s, a_2^s, \ldots, a_{s_n}^s$ *and* I *over the domains* $A_1^s, A_2^s, \ldots, A_{s_n}^s, \Sigma^*$, *respectively. In short, we will use* a^s *and* A^s *as a list* $a_1^s, a_2^s, \ldots, a_{s_n}^s$ *and a product* $A_1^s \times A_2^s \times \ldots \times A_{s_n}^s$, *respectively*;
- T *is a transition relation as in* Definition 2.1.;
- $s_0 \in S$ *is an initial state*;
- $x \in A^{s_0}$ *represents initial values for attributes* a^{s_0};
- $s_f \in S$ *is a final state.*

A tuple (s, a^s, I) is a configuration of a recognizer M. This correlates to the concept of configuration of AA in a general case. In fact, the attribute value is rewritten by its first level components. Thus, the definitions in Section 2.2. can be easily modified to specify the behaviour of a recognizer. We leave these modifications as an exercise for the reader.

Definition 2.6. *Let M be a language recognizer. The language accepted by M is the set of strings*

$$L(M) = \{w \in \Sigma^* \mid (s_0, x, w) \mapsto^* (s_f, a, \varepsilon)\}.$$

The value a is the meaning of the string w in the language $L(M)$.

Example 2.1. Figure 2 shows a language recognizer for binary numbers. In short, an enabling predicate $P(I, 'a') \equiv (head(I) = 'a')$ is represented as label **a** of the corresponding arc in the transformation graph. The symbol ♯ represents the end of the string.

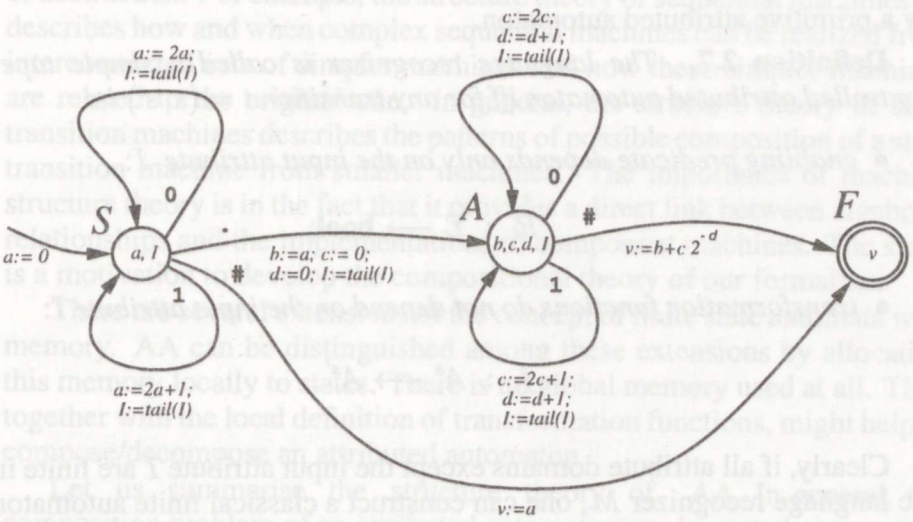


Fig. 2. Recognizer of binary numbers.

Such an automaton accepts all binary numbers with a fractional part, if necessary, and outputs their decimal values as an attribute of the final state F. This uses the well-known language of Knuth's example for attribute grammars [7], but transformed for right linear grammar.

The external (observable) behaviour of the recognizer of the language $L(M)$ can be expressed by the semantic function $\mathcal{R} : \Sigma^* \longrightarrow A^{s_f}$, implemented by the automaton M. For instance, in the last example $\mathcal{R}('1101.01') = 13.25$.

This model of the attributed language recognizer can be generalised by ignoring initialisation of attributes in the initial state. Then, these attributes are free parameters and should be given together with the input string. The recognition function will obtain the form $\mathcal{R} : A^{s_0}, \Sigma^* \longrightarrow A^{s_f}$, which defines the semantics of the accepted word, depending on the context given by the first argument.

2.4. Expressive power of attributed automata

In practice, a transformation function of an attributed automaton can be any partially recursive function. Thus, we can easily represent a Turing machine by an attributed automaton with one state and one transition, and with a Turing computable function attached to that transition. In other words, the formal power of the attributed model is that of the Turing machine. Specific cases of attributed recognizers can be compared with respect to their expressive power.

The class of the so-called primitive AA is studied in [8]. A primitive attributed automaton has natural numbers or tuples of natural numbers as attribute domains. The set of enabling predicates and transformation functions is restricted to checking whether the attribute components are zero or non-zero, and to computing succeeding and preceding natural numbers. Every partially recursive function appears to be implementable by a primitive attributed automaton.

Definition 2.7. *The language recognizer is called a simple tape-controlled attributed automaton iff for any transition $t = (s, s')$ its*

- *enabling predicate depends only on the input attribute I:*

$$P_t : \Sigma \longrightarrow \textbf{bool};$$

- *transformation functions do not depend on the input attribute I:*

$$f_t : A^s \longrightarrow A^{s'}.$$

Clearly, if all attribute domains except the input attribute I are finite in the language recognizer M, one can construct a classical finite automaton $M' = (S', T')$ with states $S' = \{(s, a^s) | s \in S, a^s \in A^s\}$ and transitions $T' = \{(c_1, c_2) | c_1 = (s_1, a^{s_1}), c_2 = (s_2, a^{s_2}), c_1 \in S', c_2 \in S', P_{(s_1,s_2)}(a) = \textbf{true}, a \in \Sigma\}$. Thus, the following is straightforward [9].

Proposition 2.8. *The simple tape-controlled AA with finite attribute domains A^s are equivalent to finite automata.*

In general terms, attributed recognizers can accept some context-sensitive languages as shown in Fig. 3. Thus, the expressive power is interesting in the context of specific cases of attributed recognizers and in that of the power of attribute domains.

3. COMPOSITIONS OF ATTRIBUTED MODELS

State transition machines are widely used in software engineering for modelling the behaviour of a system. As a means of natural graph representation, state transition machines provide intuitively understandable descriptions of small systems. Problems will appear when specification

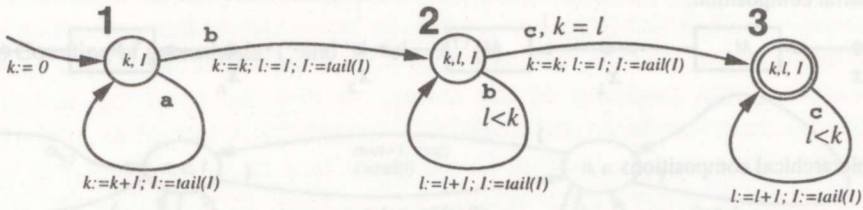


Fig. 3. Recognizer of the language $\mathcal{L} = \{a^n b^n c^n \mid n > 0\}$.

of a system consists of more than 20–30 states. Larger descriptions lose their clarity mainly because of the flat structure of state transition diagrams. To minimise this drawback, it is necessary to introduce hierarchical specifications, where machines of reasonable size are used on every level of abstraction. For example, the structure theory of sequential machines [4] describes how and when complex sequential machines can be realized from interconnected sets of simpler machines and how these simpler machines are related to the original one. In general, the structure theory of state transition machines describes the patterns of possible composition of a state transition machine from smaller machines. The importance of machine structure theory is in the fact that it provides a direct link between algebraic relationships and the implementations of component machines. The same is a motivation to develop the compositional theory of our formalism.

There are several extensions of the concept of finite state automata with memory. AA can be distinguished among these extensions by allocating this memory locally to states. There is no global memory used at all. This, together with the local definition of transformation functions, might help to compose/decompose an attributed automaton.

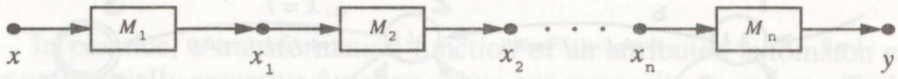
Let us summarise the structure theory of AA. In general, the composition problem of an attributed automaton can be stated as:

How and when can a complex attributed automaton be implemented by a network of simpler AA, and how are these component automata related to the automaton under consideration?

The composition studies have emphasised several specific cases of transformational AA [10–12]. There are two main compositions of transformational AA with one initial and one final state as studied previously (Fig. 4):

- a serial composition, where automata M and M' are connected by "pasting" together a final state of M and the initial state of M';
- a hierarchical composition of AA, where a transition function of an attributed automaton is implemented by another attributed automaton (or even by the same attributed automaton, in the case of direct recursion).

a) serial composition:



b) hierarchical composition:

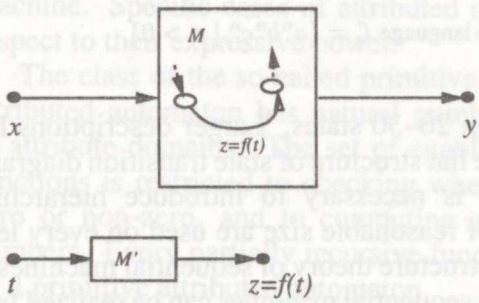


Fig. 4. Composition of attributed automata.

The hierarchical composition of simple AA can be used to specify the parsing of Dyck languages. In this case, the regular structure of the string is represented by transitions controlled by the enabling predicates, and the balance of parentheses is governed by the attributes. For every type of parentheses, one attributed automaton is used that calls another one, when another type of parentheses appears. Figure 5 shows AA for recognition where two types of parentheses are used.

In terms of software engineering, data structures must be as simple as possible. In our last example, only simple attributes (natural numbers) rather than attributes with tree structure are used to explain the balance of parentheses. Note that with such a single attributed automaton it is impossible to implement Dyck languages with more than one type of parentheses. This emphasises the importance of the composition theory of AA.

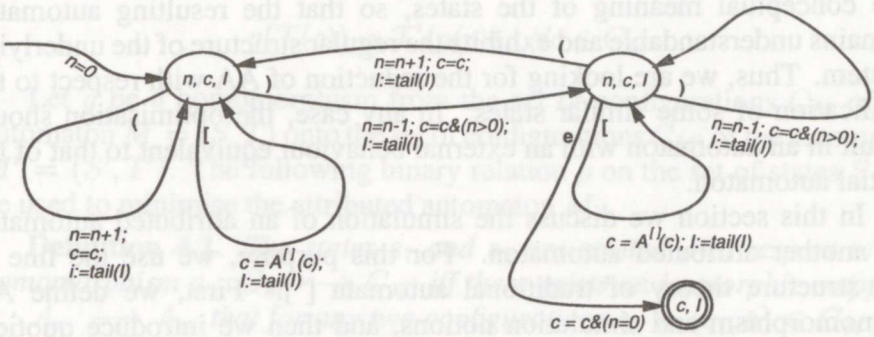
In this context, it is also significant that every CF-language is acceptable by a system of simple AA with primitive attributes. This is based on the well-known theorem [13]:

Theorem. *For every CF-language L in alphabet Σ, there exist a Dyck language L_D, a regular language R in alphabet Σ' and a homomorphism $h : \Sigma' \longrightarrow \Sigma$ such that $L = h(L_D \cap R)$.*

The serial composition and superposition of AA and the "minimisation" studied in [8] create a fundamental set of compositions for transformational AA. In addition, one can introduce a number of more complex compositions where states of automata are "pasted" together in several states. The object is to develop a concept of a mathematically well-founded general composition, which would cover all the known ones as particular

Grammar: S --> SS | ( ) | (S) | [ ] | [S]

a) Counting of parenthesis '(' and ')': $c' = A^{()}(c)$



b) Counting of parenthesis '[' and ']': $c' = A^{[]}(c)$

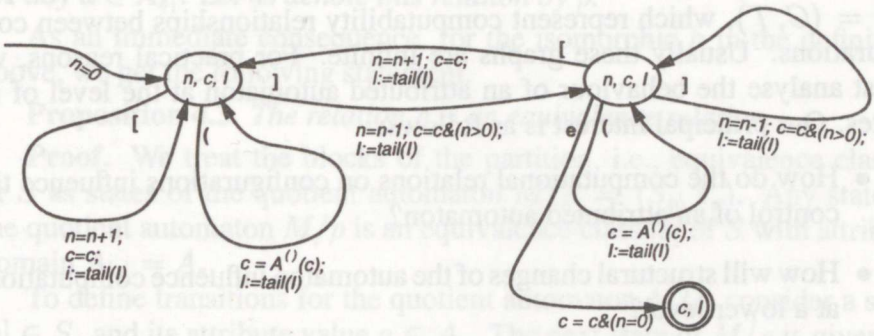


Fig. 5. Implementation of the Dyck language.

cases. Here, an algebraic theory of general composition of AA is under development. The wreath product of AA is introduced to achieve a general composition, at least for tape-controlled simple AA [9].

For interactive AA, some other types of compositions can be used, such as parallel compositions with different kinds of synchronisation, disjoint union, external and internal choice, action hiding, relabeling, and their timed equivalents (for real-time applications). A preliminary version of the composition of interactive AA is presented in [12].

4. SIMULATION OF ATTRIBUTED AUTOMATA

Another way to keep an automaton at a reasonable size is to minimise its number of states. It is always possible to compose some succeeding transitions (together with intermediate states) into one by analytical transformations of the predicates and functions involved. The ultimate goal is to reduce the structure of an attributed automaton globally into one

state and one reflexive transition. On the other hand, this makes efficient analytical transformations less feasible or non-existent in a general case. In the engineering context, the reduction of the automata should not destroy the conceptual meaning of the states, so that the resulting automaton remains understandable and exhibits the regular structure of the underlying system. Thus, we are looking for the reduction of AA with respect to the unification of some similar states. In any case, the optimisation should result in an automaton with an external behaviour equivalent to that of the initial automaton.

In this section we discuss the simulation of an attributed automaton by another attributed automaton. For this purpose, we use the line of the structure theory of traditional automata [4]. First, we define AA homomorphism and simulation notions, and then we introduce quotient automata. On this theoretical basis, we can develop the composition methods of AA, by creating minimisation algorithms. For simplicity, here we describe only deterministic AA.

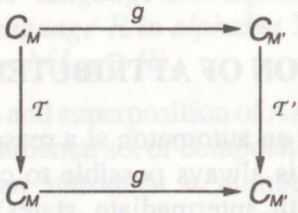
Eventually, the behaviour of AA can be followed on data-flow graphs $\mathcal{M} = (C, \mathcal{T})$, which represent computability relationships between configurations. Usually these graphs are infinite. For practical reasons, we must analyse the behaviour of an attributed automaton at the level of its states. Our principal interest is as follows:

- How do the computational relations on configurations influence the control of an attributed automaton?
- How will structural changes of the automaton influence computations at a lower level?

In our comparison of AA at the level of the structure of automata, we discuss behavioural homomorphism (e.g., isomorphism and equivalence) and whether the corresponding relations between data-flow graphs of these automata exist.

4.1. Internal view

Definition 4.1. *Let $M = (S, T)$ and $M' = (S', T')$ be deterministic AA. Attributed automaton M is behaviourally homomorphic to attributed automaton M' iff there is a homomorphic function $g : C_M \longrightarrow C_{M'}$ that makes the following diagram commutative*



AA M and M' are behaviourally isomorphic if the homomorphism $g : C_M \longrightarrow C_{M'}$ is a bijection.

In other words, there is a behavioural homomorphism from the attributed automaton M to M' if there is homomorphism between their data-flow graphs:

$$g(\mathcal{T}(c)) = \mathcal{T}'(g(c)), \ \forall c \in C_M.$$

Let g be a homomorphism from the set of configurations C_M of the automaton $M = (S, T)$ onto the set of configurations $C_{M'}$ of the automaton $M' = (S', T')$. The following binary relation ρ on the set of states S can be used to minimise the attributed automaton M.

Definition 4.2. *The states s_1 and s_2 are related with respect to the homomorphism $g : C_M \longrightarrow C_{M'}$ iff there exists an isomorphic mapping $h : A_{s_1} \longrightarrow A_{s_2}$ that for any two configurations $c_1 = (s_1, a) \in C_M$ and $c_2 = (s_2, h(a)) \in C_M$, the homomorphism g gives the same value:*

$$g(c_1) = g(c_2)$$

for any $a \in A_{s_1}$. Let us denote this relation by ρ.

As an immediate consequence, for the isomorphic h in the definition above, we get the following statement.

Proposition 4.3. *The relation ρ is an equivalence relation.*

Proof. We treat the blocks of the partition, i.e., equivalence classes of S as states of the quotient automaton $M/\rho = (S_\rho, T_\rho)$. Any state of the quotient automaton M/ρ is an equivalence class $[s]$ of S with attribute domain $A_{[s]} = A_s$.

To define transitions for the quotient automaton M/ρ, consider a state $[s] \in S_\rho$ and its attribute value $a \in A_s$. The next state of M/ρ is given by $[\tau(s, a)]$, and its attribute is given by $\alpha(s, a)$. Here, τ and α are projections of the data-flow relation $\mathcal{T}$. In the following, we denote by τ' and α' the same projections of the data-flow relation $\mathcal{T}'$ for the automaton M'.

When we classify the states of an automaton, we should verify against Def. 2.1. whether the introduced transitions are predicates and attribute transformations functions. This can be applied to our model as follows:

Proposition 4.4. *The quotient automaton M/ρ is well defined.*

Proof. Any transition $([s_1], [s'])$ of the quotient automaton M/ρ is *independent* of the choice of the class representative $[s]$. Suppose that $s_2 \in [s_1]$ for some configurations $c_1 = (s_1, a) \in C_M$, $c_2 = (s_2, h(a)) \in C_M$, where an isomorphism h maps A_{s_1} to A_{s_2} and for any $a \in A_{s_1}$: $g(c_1) = g(c_2)$. According to the homomorphism g between C_M and $C_{M'}$ we also have

$$g(\tau(s_1, a), \alpha(s_1, a)) = (\tau'(g(s_1, a)), \alpha'(g(s_1, a))),$$

$$g(\tau(s_2, h(a)), \alpha(s_2, h(a))) = (\tau'(g(s_2, h(a))), \alpha'(g(s_2, h(a)))),$$

$$g(\tau(s_1, a), \alpha(s_1, a)) = g(\tau(s_2, h(a)), \alpha(s_2, h(a))),$$

i.e., the next configurations in C_M for states s_1 and s_2 with the same attribute value a are mapped by g into the same configuration in $C_{M'}$, thus

they belong to the same class $[s_k]$ on S of M. Hence, the transition function T_ρ of the attributed automaton M/ρ is well defined.

Let M and M' be AA the behaviour of which is similar with respect to the homomorphism $g : C_M \longrightarrow C_{M'}$, where C_M and $C_{M'}$ are sets of configurations of the machine M and M', respectively. The following theorem states that in this case the behaviour is isomorphic to the behaviour of the quotient automaton M/ρ.

Theorem 4.5. *Let $g : C_M \longrightarrow C_{M'}$ be a homomorphism for deterministic AA M and M'. Then M' is behaviourally isomorphic to the quotient automaton M/ρ.*

Proof. For automata M/ρ and M', let us introduce a mapping $f : C_{M/\rho} \longrightarrow C_{M'}$ such that $f([s], a) = g(s, a)$, for any $c = (s, a) \in C_M$. The mapping f is well defined due to the properties of g considered above. For f to be an isomorphism, it must be a bijection and a homomorphism. Let $c' = (s', a') \in C_{M'}$. Then $c' = g(c)$ for some $c \in C_M$, where $c = (s, a)$. The automaton M/ρ has a state $[s]$, and it contains $f([s], a) = c'$. Consequently, f is a mapping onto.

To demonstrate that f is one-to-one, suppose $f([s_k], a) = f([s_l], a')$ for some $a \in A_{s_k}$ and $a' \in A_{s_l}$. Then $g(s_k, a) = g(s_l, a')$ and, therefore, states s_k and s_l are in the same class $[s_k] = [s_l]$.

Now let us consider transitions, starting with the configuration $c = (s, a) \in C_M$. The next configuration for $([s], a)$ in M/ρ is defined for any $a \in A_{[s]}$ as the pair $([\tau(c)], \alpha(c))$. The next configuration for $f([s], a) = g(c)$ is $(\tau'(g(c)), \alpha'(g(c)))$. Moreover,

$$f([\tau(c)], \alpha(c)) = g(\tau(c), \alpha(c)) = (\tau'(g(c)), \alpha'(g(c))).$$

Consequently, the corresponding configurations proceed under f into corresponding configurations, and therefore f defines a behavioural isomorphism from the automaton M/ρ to the attributed automaton M'.

4.2. External view

Next, we focus on the external behaviour of AA, i.e., we only consider the evaluated attribute values or observed computations and ignore the homomorphism of the control structures of automata. Again, let us consider a deterministic automaton $M = (S, T)$ and its quotient automaton $M' = (S', T') \equiv M/\rho$. Let $\mathcal{F}$ denote the mapping that transforms M to M'. Theorem 4.5. shows that AA M and M' have (because of isomorphisms between attributes of equivalent states of the automaton M) equivalent internal behaviour. This is illustrated in Fig. 6, where a fragment of dependencies between attribute values is represented. The nodes $a_0, a_2, \ldots$ present different attribute values. Attribute domains are encircled by ovals labelled by the names of the corresponding states. The structures of both an attributed automaton M

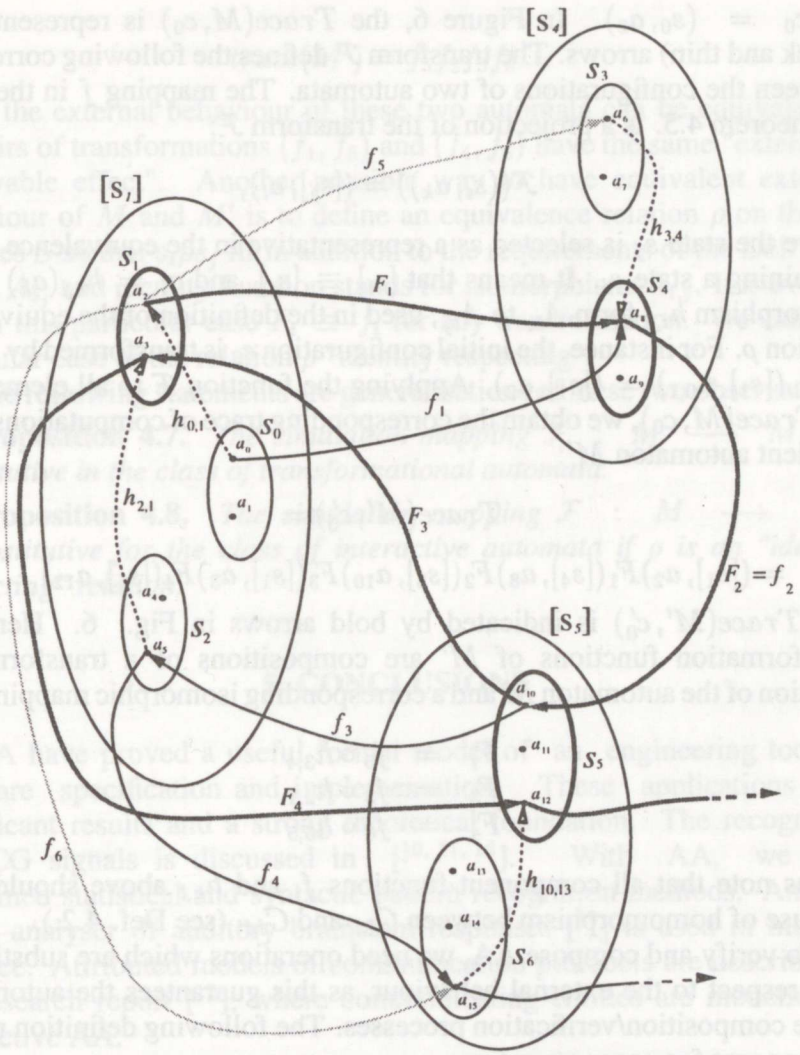


Fig. 6. Data-flow of the structured attributed automaton.

and its quotient automaton M' overlap. An initial automaton has states labelled by $s_0, s_1, \ldots$, whereas the labels of states of the quotient automata are $[s_1], [s_4], \ldots$. Thus, an automaton M has possible configurations $C_M = \{(s_0, a_0), (s_0, a_1), (s_1, a_2), (s_1, a_3), (s_3, a_4), \ldots, (s_6, a_{15})\}$. The quotient automaton has the set of configurations $C_{M'} = \{([s_1], a_2), ([s_1], a_3), ([s_4], a_8), ([s_4], a_9), ([s_5], a_{10}), ([s_5], a_{11}), ([s_5], a_{12})\}$. In Fig. 6, transformation functions of the automaton M are labelled by $f_1, f_2, \ldots$ and transformation functions of its quotient automaton M' by $F_1, F_2, \ldots$.

Let us show an example of an internal trace of computations of the automaton M:

$$Trace(M, c_0) = (s_0, a_0) f_1 (s_4, a_8) f_2 (s_5, a_{10}) f_3 (s_2, a_5) f_4 (s_6, a_{15}) \ldots$$

for $c_0 = (s_0, a_0)$. In Figure 6, the $Trace(M, c_0)$ is represented by (black and thin) arrows. The transform $\mathcal{F}$ defines the following correlation between the configurations of two automata. The mapping f in the proof of Theorem 4.5. is a projection of the transform $\mathcal{F}$:

$$\mathcal{F}((s_i, a_k)) = ([s_j], a_l),$$

where the state s_j is selected as a representative in the equivalence block, containing a state s_i. It means that $[s_i] \equiv [s_j]$, and $a_l = h_{i,j}(a_k)$ for an isomorphism $h_{i,j}$ form A_{s_i} to A_{s_j}, used in the definition of the equivalence relation ρ. For instance, the initial configuration c_0 is transformed by $\mathcal{F}$ into $c_0' = ([s_1], h_{0,1}) = ([s_1], a_2)$. Applying the function $\mathcal{F}$ to all elements of the $Trace(M, c_0)$, we obtain the corresponding trace of computations at the quotient automaton M':

$$Trace(M', c_0') =$$

$$= ([s_1], a_2)F_1([s_4], a_8)F_2([s_5], a_{10})F_3([s_1], a_3)F_4([s_5], a_{12}) \ldots$$

The $Trace(M', c_0')$ is indicated by bold arrows in Fig. 6. Here, the transformation functions of M' are compositions of a transformation function of the automaton M and a corresponding isomorphic mapping $h_{i,j}$:

$$\begin{aligned} F_1 &= f_5 \circ h_{3,4} \\ F_3 &= f_3 \circ h_{2,1} \\ F_4 &= f_6 \circ h_{6,5} \end{aligned}$$

Let us note that all component functions f_i and $h_{k,l}$ above should exist because of homomorphism between C_M and $C_{M'}$ (see Def. 4.2.).

To verify and compose AA, we need operations which are substitutive with respect to the external behaviour, as this guarantees the automation of the composition/verification processes. The following definition refines this concept for unary operations.

Definition 4.6. *The operation F over a class of AA $\mathcal{A}$ is substitutive if for any automaton $M \in \mathcal{A}$ and any (external) execution trace $trace(M, c_0)$ the following is valid*

$$F(trace(M, c_0)) \Longrightarrow trace(F(M, c_0)).$$

In the formal example above, we can see that the simulating operation $\mathcal{F}$ is substitutive for transformational AA up to the isomorphisms $h_{k,l}$. The second components of the configurations in the traces $Trace(M, c_0)$ and $Trace(M', c_0')$ are isomorphic, and so, as well for final configurations. That means that if a deterministic attributed automaton M computes y from x, then $M'(x') = y'$ is valid for some isomorphic mappings $h_1(x) = x'$ and $h_2(y') = y$.

The situation is different for transformational automata. Let us define all transformations $f_1, f_2, \ldots$ to be observable. Then

$$trace(M) = f_1 f_2 f_3 f_4 \ldots$$

and

$$trace(M') = f_5 f_2 f_3 f_6 \ldots.$$

Thus, the external behaviour of these two automata can be equivalent if the pairs of transformations (f_1, f_5) and (f_4, f_6) have the same "externally observable effect". Another possible way to have equivalent external behaviour of M and M' is to define an equivalence relation ρ on the set of states S so that $s_i \rho s_j$ iff in addition to the requirements of the Def. 4.2., $A_{s_i} = A_{s_j}$ and identity function stands for isomorphisms $h_{i,j}$. It is evident that in this particular case $F_i \equiv f_i$ for any transformation. We call this particular case of the relation ρ "identity respecting".

The following statements are generalisations of these two observations.

Proposition 4.7. *The simulation mapping $\mathcal{F} : M \longrightarrow M/\rho$ is substitutive in the class of transformational automata.*

Proposition 4.8. *The simulation mapping $\mathcal{F} : M \longrightarrow M/\rho$ is substitutive for the class of interactive automata if ρ is an "identity respecting" relation.*

5. CONCLUSIONS

AA have proved a useful formal model of an engineering tool for software specification and implementation. These applications have significant results and a strong theoretical foundation. The recognition of ECG signals is discussed in [10, 11, 14]. With AA, we have combined statistical and syntactic pattern recognition methods. An AA-based analyser of auditory brainstem responses [15] is used in medical practice. Attributed models of communication protocols are described in the research report [12], where communicating entities are modelled by interactive AA.

The applications above indicated the following research problems:

- specification language of AA, designed for some particular problem domains (for biomedical applications – in [10, 14], for protocol specification applications – in [12]);
- tuning methods for AA (learning of AA);
- composition/decomposition methods and tools for AA.

In addition to the application areas mentioned, the same formal model could be successfully used in natural language processing (e.g., compare the method of ATN in [16]) and graph-based problems (e.g., modelling of flows in public traffic, routing of mobile phones). It seems reasonable to develop a proper concept of AA as an engineering tool. In the report [12], the first attempts at this were made by means of the NUT system. On the one hand, this permits a comparison with other formalisms (in particular, with Tyugu's computational frames [17]). On the other hand, AA is a

simple and useful means to organise one's ideas when evaluating complex problems. For engineering, the graphical implementation environment needs to be developed as well.

With a suitable environment for the specification of AA, the simulation of several practical systems is straightforward. We need to develop additional regular structures as the basis for AA. In all applications, this abstraction still has to be made by a human. Obviously, in many cases, it will be the most powerful method. It is worth investigating AA in the context of machine learning of AA and in the context of generative programming.

ACKNOWLEDGEMENTS

The authors thank U. Kaljulaid, E. Tyugu, T. Grönfors, and A. Koski for cooperation and useful discussions on the topic.

REFERENCES

1. Schulz, K. U. and Gabbay, D. M. Logic finite automata. – In: Pólos, L. and Masuch, M. (eds.). Applied Logic: How, What, and Why? Kluwer Academic Publishers, Boston, 1995, 237–287.
2. Halbwachs, N. Delay analysis in synchronous programs. – In: Lecture Notes in Comp. Sci., **697**. Springer-Verlag, 1993, 336–346.
3. Liu, M. T., Jeng, H.-W. J., and Koch, L. S. Formal description techniques for protocol specification. – In: Proc. of the ATR International Workshop on Communications Software Engineering, Kioto, Oct. 20–21, 1994. Kioto, 1994, 31–71.
4. Hartmanis, J. and Stearns, R. E. Algebraic Structure Theory of Sequential Machines. Prentice Hall, Englewood Cliffs, New York, 1966.
5. Meriste, M. Attributed Automata: Some Results and Open Problems. Res. Rep. CS75/94, Inst. of Cybernetics, Estonian Academy of Sciences, Estonia and Dep. of Comp. Sci., University of Tartu. Tallinn, 1994.
6. Meriste, M. and Penjam, J. On Formal Models of Finite Attributed Automata. Res. Rep. CS52/92, Inst. of Cybernetics, Estonian Academy of Sciences, Estonia and Dep. of Comp. Sci., University of Turku, Finland. Tallinn, 1992.
7. Knuth, D. E. Semantics of context-free languages. – Math. Systems Theory, 1968, **2**, 127–145.
8. Meriste, M. and Vene, V. Attributed automata and language recognizers. – In: Proc. of the Fourth Symposium on Programming Languages and Software Tools. Visegrád, Hungary, June 9–10, 1995. Department of General Computer Science, Eötvös Lórand University, Budapest, 1995, 114–121.
9. Kaljulaid, U., Meriste, M., and Penjam, J. Algebraic Theory of Tape-Controlled Attributed Automata. Res. Rep. CS59/93, Inst. of Cybernetics, Estonian Academy of Sciences, Estonia. Tallinn, 1993.
10. Juhola, M., Koski, A., and Meriste, M. Structural recognition of one-dimensional patterns. – In: Proc. of the Conference on Artificial Intelligence Research in Finland, Turku, Finland, August 1994. Turku University, Turku, 1994, 95–108.
11. Juhola, M., Koski, A., and Meriste, M. Syntactic Recognition of ECG Signals by Attributed Finite Automata. (to appear in *Pattern Recognition*).
12. Penjam, J. Attributed Automata: A Formal Model for Protocol Specification. Res. Rep. TRITA-IT-94:30, IOC-CS-71/94. Royal Institute of Technology and Institute of Cybernetics. Stockholm, 1994.

13. Ginsburg, C. and Greibach, S. Deterministic context-free languages. – Inform. and Control, 1966, **9**, 6, 620–648.
14. Koski, A. On Structural Recognition Methods Applied to ECG Analysis. Licenciate thesis. Res. Rep. R-94-11, Dep. of Comp. Sci., University of Turku, Finland, May 1994.
15. Grönfors, T. Novel Methods of Syntactic Pattern Recognition for Peak Detection of Auditory Brainstem Responses. Doctoral dissertation. Dep. of Comp. Sci. and Appl. Math., University of Kuopio, Finland, Nov. 1994, 137.
16. Blank, G. D. A finite and real-time processor for natural language. – Comm. ACM, 1989, **32**, 10, 1174–1189.
17. Tyugu, E. Knowledge-Based Programming. Addison-Wesley, New York, München, 1988.

ARVUTUSTE ATRIBUUTMUDELID

Merik MERISTE, Jaan PENJAM

On käsitletud teadmiste ja arvutuste esitamise uut formaalset mudelit – atribuutautomaati (AA). AA on lõpliku automaadi üldistus, kus automaadi iga olekuga seotakse lõplik hulk muutujaid e. atribuute ja iga ülemineku kaasnevad semantilised teisendused. AA võimaldab kirjeldada kontseptuaalseid teadmisi mõistete regulaarse süntaktilise struktuuri baasil, atribuute kasutatakse mõistete kontekstist sõltuvate omaduste ja tähenduste spetsifitseerimiseks. AA mudel võimaldab restruktureerida keeruliste süsteemide efektiivselt realiseeritavaid spetsifikatsioone ja vähendada nende kontseptuaalset keerukust. On vaadeldud AA üldisi teoreetilisi probleeme ning AA kompositsiooni/dekompositsiooni ja optimeerimise küsimusi.

АТРИБУТНЫЕ МОДЕЛИ ВЫЧИСЛЕНИЙ

Мерик МЕРИСТЕ, Яан ПЕНЬЯМ

Рассмотрены атрибутные автоматы – новая формальная модель представления знаний, основывающаяся на их атрибутированном регулярном синтаксисе. Атрибутами представляются контекстуальные свойства и семантика описываемых концепций. Атрибутный автомат является обобщением конечного автомата, состояниям автомата приписываются атрибуты – переменные и переходам – семантические действия. Атрибутный автомат как формальная модель предлагает дополнительные средства для реструктурирования больших систем, способствуя тем самым упрощению концептуальной сложности, спецификации и повышению эффективности реализации. Рассмотрены общие теоретические проблемы атрибутных автоматов, а также вопросы их композиции/декомпозиции и оптимизации.