INFORMATICA, 1998, Vol. 9, No. 1, 85–105
© 1998 *Institute of Mathematics and Informatics, Vilnius*

Models of Attributed Automata *

Merik MERISTE
Centre of Technology, University of Tartu, Liivi 2, EE2484 Tartu, Estonia
e-mail: merik@ut.ee

Jaan PENJAM
Institute of Cybernetics, Akadeemia tee 21, EE0026 Tallinn, Estonia
e-mail: jaan@cs.ioc.ee

Varmo VENE
Institute of Computer Science, University of Tartu, Liivi 2, EE2484 Tartu, Estonia
e-mail: varmo@cs.ut.ee

Received: January 1998

Abstract. Attributed automaton (AA) is a formalism for conceptual knowledge specification using regular syntax with attributes representing contextual relations as well as semantic properties of concepts. AA can be treated as a generalization of a finite automaton with attributes and computational relations attached to states and transitions respectively. In this paper we develop a new specification method for AA based on functional combinators. It allows modular specification of AA, enjoys good algebraic properties and is extendeble for different kind of attributed automata.

Key words: attributed automata, language recognizers, functional specification 2 of AA, functional parsers.

1. Introduction

Regular and context-free structure are classical and efficiently implementable models of data structures. Automated computing is often as successful as adequately regular and/or context-free (surface or deep) substructures are extracted from the rest of structure of data. Formal models integrating these structures with others and supporting data restructuring, analysis and implementation are methodologically important. The modifications of models known in the theory of formal languages and automata (e.g., finite and push-down automata and state transition systems), and corresponding declarative formalisms (e.g., attributed grammars, graph grammars) are widely used to achive effectively executable specifications.

Attributed automata (AA) were introduced in (Meriste and Penjam, 1992a) as a formalism for executable specifications using regular syntax with attributes representing

*Partially supported by Estonian Science Foundation grant ETF 1718.

contextual relations as well as semantic properties of underlying concepts. Due to applications in medical signal analysis (Grönfors and Meriste, 1992; Juhola and Meriste, 1992) attributed automata is accepted as a software engineering tool.

From the theoretical aspects AA have been investigated by means of rewriting systems (Meriste and Penjam, 1992b) and in terms of classical automata theory (Meriste, 1994; Meriste and Penjam, 1995a; Meriste and Penjam, 1995b). The algebraic theory of (de)composition of some classes of AA is developed in (Kaljulaid *et al.*, 1993). However, several problems remained still open. Particulary, there was no general theory of composition of AA and good specification methodology.

In this paper, we develop a functional specification method for attributed automata, which has good compositional properties. We will proceed as follows. In the next section we give a short overview of the formal model of attributed automata. In Section 3 we develop a new functional specification method for attributed automata. Then, in Sections 4 and 5, we study the properties of automata specified functionally and show how the method can be naturally extended for the more generel attributed automata. In Section 6, we compare our method with functional parsers. Finally, in Section 7, we conclude with discussion about open problems and future work.

2. Finite Attributed Automata

DEFINITION 1. A finite attributed automaton M is a tuple $M = (I, S, A, \sigma, s_0, s_F)$, where

(a) I is a finite alphabet of terminal symbols,

(b) S is a finite alphabet of states,

(c) $A = \{A_s \mid s \in S\}$ is a family of domains of attributes indexed over states,

(d) $\sigma = \{\sigma_{ss'} \subseteq (A_s \times I^*) \times A_{s'} \mid s, s' \in S\}$ is a family of transition relations indexed over pairs of states,

(e) $s_0 \in S$ is an initial state and

(f) $s_F \in S$ is a final state.

Starting in the initial state s_0 with the initial attribute value $a_0 \in A_{s_0}$ and input string $w \in I^*$ the functioning of the automaton M is considered as successive change of the current state. The transition from the state s to s' is possible only if $(a, u)\sigma_{ss'}(a')$, where $a \in A_s$, $a' \in A_{s'}$ are corresponding attributes and $u \in I^*$ is the prefix of the input string to be read next. Automaton terminates if there is no transition possible. If this happens when the current state $s \in S$ is final (i.e., $s = s_F$), then automaton terminates with success. The remaining input string w' together with final attribute value $a \in A_{s_F}$ form a result of automaton M. If the automaton stops in the state which is not final, then the automaton fails. It is also possible that automaton will not terminate at all, as in the case

of infinite loop or if the computation of the transition relation doesn't terminate for given arguments. In both cases the result of automaton is undefined.

DEFINITION 2. A *configuration* of the attributed automaton $M = (I, S, A, \sigma, s_0, s_F)$ is a triple

$$(s, a, w) \in (S \times A_s \times I^*),$$

where

(a) $s \in S$ is the current state of the attributed automaton M,

(b) $a \in A_s$ is the attribute value of the current state s,

(c) $w \in I^*$ represents the unused portion of the input string. If $w = \varepsilon$, where ε stands for the empty string, it is assumed that the whole input string has been read.

The *initial configuration* of the automaton M is a configuration of the form

$$(s_0, a_0, w),\ w \in I^*.$$

The *final configuration* of the automaton M is a configuration of the form

$$(s_F, a, w),\ a \in A_{s_F},\ w \in I^*,$$

such that there is no transition possible.

Now, the automaton M can be considered as a formal computing device which operating cycle is running over configurations.

DEFINITION 3. The *move* by M is a binary relation $\mapsto_M$ (or simply $\mapsto$) on configurations as follows

$$(s, a, ww') \mapsto_M (s', a', w') \iff (a, w)\sigma_{ss'}(a'),$$

where $s, s' \in S$ are states, $a \in A_s$, $a' \in A_{s'}$ are corresponding attributes, $w \in I^*$ is the consumed prefix of the input string and $w' \in I^*$ is the remaining input string. It's reflexive transitive closure is denoted by $\mapsto_M^*$.

Note, that during a move, the prefix with arbitrary length can be consumed from the input string. The particular case, when no symbols are consumed, is called the ε-*move.* In fact, there no principal need to allow more than one symbol to be consumed during a move. Consuming longer prefix can be easily simulated by the sequence of moves consuming only one symbol. On the other hand, the ε-move can be not so easily simulated. It makes possible the automaton to loop forever. If ε-move are not allowed, every move consumes some (non empty) prefix of the input string, and the only reason for non-termination of the automaton can be the non-terminating transition relation or infinite input string.

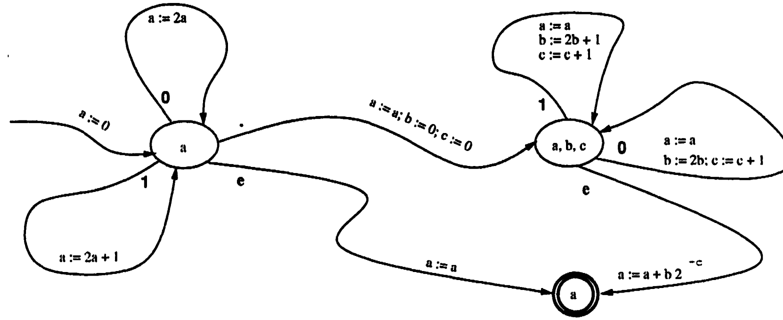


Fig. 1. Recognizer of the binary numbers.

PROPOSITION 1. Let all the transition relations $\sigma_{ss'}$ be total and such that induced move relation doesn't allow ε-moves (i.e., $\sigma_{ss'} \subseteq (A_s \times I^+) \times A_{s'}$). Then for every finite input string $w \in I^*$ and for every initial attribute value $a_0 \in A_{s_0}$, the attributed automaton $M = (I, S, A, \sigma, s_0, s_F)$ is guaranteed to terminate.

In general, for a given configuration (s, a, w), there can be several configurations related with it by the move relation.

DEFINITION 4. A finite attributed automaton $M = (I, S, A, \sigma, s_0, s_F)$ is *deterministic* iff for every configuration (s, a, vw) there exists at most one configuration (s', a', w) such that $(s, a, vw) \mapsto_M (s', a', w)$, i.e., the move relation $\mapsto_M$ is a function. Otherwise the automaton M is *non-determinictic*.

Deterministic attributed automata are interesting from the applications point of view, as their efficient implementation is much easier than non-deterministic ones. On the other hand, some problems can be specified more naturally using non-deterministic automata. The situation is similar to the finite automata, where regular expressions correspond directly to the non-deterministic finite automata and these are transformed to deterministic ones. Unfortunately, because of free domains of attributes, the corresponding transformation of attributed automata is impossible in general case (or otherwise we are able to solve the halting problem).

It is often illustrative to present AA in the form of transition graphs as in the Fig. 1. States of the attributed automaton are represented by nodes labelled by the associated attribute or by components of the attribute if the attribute is a tuple. Every arc from a state s to a state s' is labelled by a triple

$$(w, P, f) \;:\; I^* \times (A_s \to \mathbf{bool}) \times (A_s \to A_{s'}),$$

where s and s' are states corresponding to the source and the target node respectively, P is an *enabling predicate* and f is a *transformation function*. The set of arcs from the

state s to the state s' with labels $(w_1, P_1, f_1), \ldots, (w_n, P_n, f_n)$ represents the transition relation $\sigma_{ss'}$ in the following sense:

$$(a, w)\sigma_{ss'}(a') \iff \exists i \;\; 1 \leqslant i \leqslant n \quad w = w_i \,\wedge\, P_i(a) = \mathbf{tt} \,\wedge\, a' = f(a)\,.$$

Note that in the case of non-deterministic automata it is not always possible to represent the transition relation $\sigma_{ss'}$ by a finite set of arcs.

2.1. *Attributed Automata as Recognizers*

Language recognition is a natural application of attributed automata. By specializing the general definition of finite attributed automata we can easily obtain classical language recognition devices. For instance:

- The attributed automaton without attributes is a classical *finite automaton.*
- The attributed automaton with stack as it's only attribute in every state is a *push-down automaton.*

Below we consider two different approaches of using attributed automata as language recognizers. First, we consider a string to be *recognized,* if the analysis of that string ends with the final configuration where the whole input string is consumed. In the second approach we introduce predicates on final attribute values to represent contextual requirements for the acceptable string. We will introduce both notions of the accepted language as follows.

DEFINITION 5. Let $M = (I, S, A, \sigma, s_0, s_F)$ be an attributed automaton. The *language* accepted by the automaton M is defined as follows

$$L(M, a_0) = \{w \in I^* \mid (s_0, a_0, w) \mapsto^*_M (s_F, a, \varepsilon),\; a \in A_{s_F}\}.$$

The final attribute value $a \in A_{s_F}$ can be interpreted as the *meaning* of the string w in the language $L(M, a_0)$. For example, the attributed automaton from the Fig. 1 recognizes binary numbers and the final attribute value a represents the decimal value of the binary number. For instance if the input string is $w =$ "1101.01″ then the final attribute value is $a = 13.25$.

As an another example, consider the language $\mathcal{L} = \{a^n b^n c^n \mid n \geqslant 0\}$ which is context-sensitive. A recognizer for the language $\mathcal{L}$ is the automaton given in Fig. 2. In this example attributes play a different role – they collect contextual information which is used on some states to decide which transition to choose.

Because of domains of attributes can be infinite and transition relation can be arbitrary computable relation, the expressive power of attributed automata are that of Turing machines. On the other hand, we often don't use the full generality of attributed automata. As we see in our examples, in most cases attributes are used to collect some information, which is used only on some special places for deciding the transition to choose. In

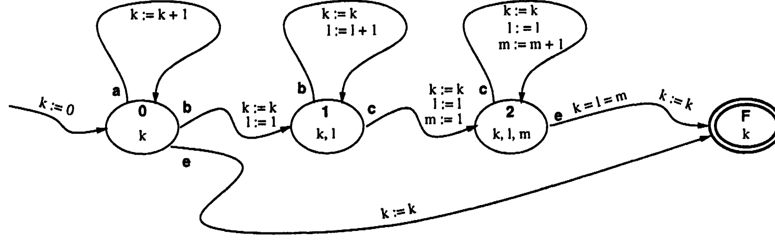


Fig. 2. Recognizer of the language $\mathcal{L} = \{a^n b^n c^n \mid n \geqslant 0\}$.

fact, the automaton in Fig. 1 don't use attributes at all for the recognition and the automaton in Fig. 2 uses them only on the last transition. This gives idea for the following decomposition of attributed language recognizer.

DEFINITION 6. An attributed automaton $M = (I, S, A, \sigma, s_0, s_F)$ is *simple* if all its enabling predicates are constantly true.

In our examples the first automaton is simple, but the second isn't. We can make it easily to the simple one by taking the sub-automaton without the final state and transition to it (and declaring the state **2** to be final).

Clearly, by means of Definition 5, simple attributed automata can recognize exactly the class of regular languages. It's because they can collect arbitrary contextual information (and compute arbitrary meaning to the string), but they can't use this information for the recognition. To make the use of collected information, we use the additional predicate to tell whether the string belongs to the language or not.

DEFINITION 7. Let $M = (I, S, A, \sigma, s_0, s_F)$ be a simple attributed automaton, $a_0 \in A_{s_0}$ the initial attribute value and $\mathcal{P} : A_{s_F} \longrightarrow$ **bool** a computable predicate on final attribute. We define the *language* accepted by the automaton M and the *accepting predicate* $\mathcal{P}$ as

$$L(M, \mathcal{P}) = \{w \in I^* \mid (s_0, a_0, w) \mapsto^*_M (s_F, a, \varepsilon),\ \mathcal{P}(a),\ a \in A_{s_F}\}.$$

From languages point of view, the initial attribute $a_0 \in A_{s_0}$ represents the *initial context* and the predicate $\mathcal{P}$ *accepting context.*

Now, the recognizer for the language $\mathcal{L} = \{a^n b^n c^n \mid n \geqq 0\}$ can be constructed by taking M to be the simple sub-automaton from Fig. 2 (as described above) and the accepting predicate P to be the enabling predicate from the last transition. This shows that at least some context-sensitive languages can be recognized by a simple attributed automaton, which just counts the occurences of certain substrings, together with an accepting predicate which is also very simple (checking the equality of natural numbers). Because, in general, the accepting predicate $\mathcal{P}$ can be any computable predicate, the class of languages recognizable by a simple automaton M and a predicate $\mathcal{P}$ is the class of recursively enumerable languages. However, this doesn't mean that all recursively enumerated languages can be recognized as easily as in the $\mathcal{L} = \{a^n b^n c^n \mid n \geqq 0\}$ case. For

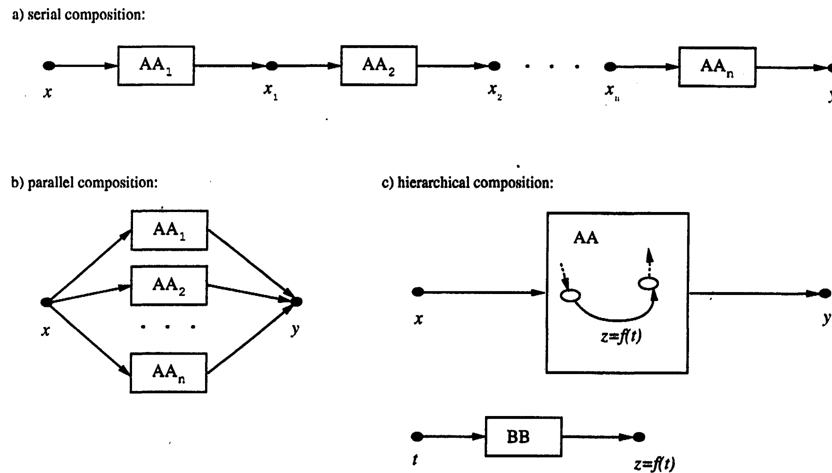


Fig. 3. Compositions of attributed automata.

instance, it is impossible to recognize Dyck languages (i.e., the languages of balanced parenthesis) with more than one type of parenthesis by a simple attributed automaton and an accepting predicate similar to the previous example. In fact, to recognize a Dyck language, the automaton has to simulate the stack as in the classical push-down automata.

As we see in our examples, we have two alternative possibilities to construct a recognizer on the basis of an attributed automaton. First, we can specify predicates at transitions, i.e., we select the next move in accordance with context conditions. Second, we can simply collect the contextual information we need to some state and specify by a predicate the acceptable context at that state. The second approach appears to be interesting from the methodological point of view, as it suggests more systematic and modular way for building language recognizers.

2.2. *Compositions of Attributed Automata*

The Definition 7 decomposed the problem of language recognition into two parts: the syntactic recognition of regular structures by a simple attributed automaton and the analysis of contextual dependences by an accepting predicate. Because the accepting predicate can be viewed as an attributed automaton with one transition labelled by the predicate, we have the (sequential) composition of two automata.

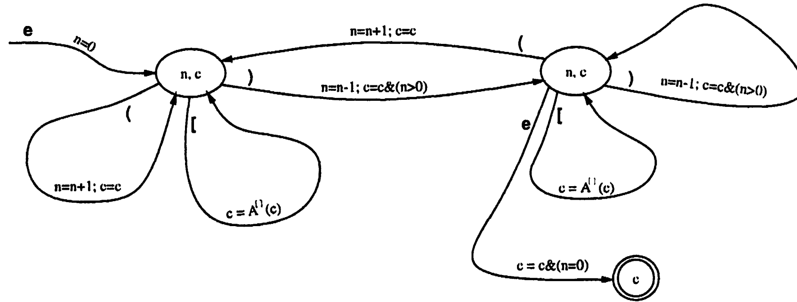
In general, the composition problem of attributed automata is: *how and when a complex attributed automaton can be simulated by interconnected sets of simpler attributed automata; how these component automata are related to the automaton under consideration?*

There are three different types of composition of attributed automata into the system that have been studied previously (Fig. 3):

- *a sequential composition* (a serial composition), where two automata are connected by "pasting" together the final state of the first automaton and the initial

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

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



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

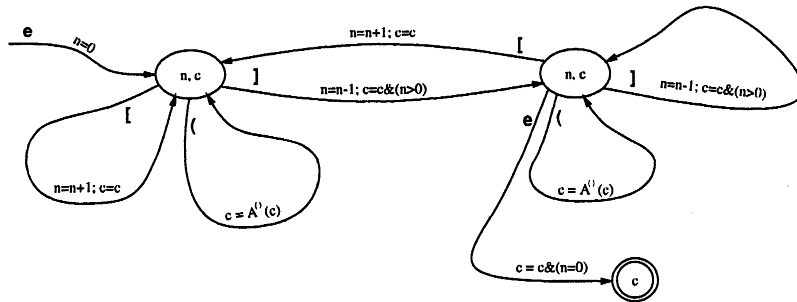


Fig. 4. The implementation of the Dyck language.

state of the second one;

- *a parallel composition*, where two automata are connected by "pasting" together their initial states and final states;
- *a hierarchical composition*, where during the transition the another attributed automaton is called (or even the same automaton in the case of direct recursion).

In fact, the sequential and the parallel composition are both special cases of the hierarchical composition. In the sequential case, we can add a new final state to the first automaton together with the transition from the old final state and call the second automaton at this transition. It is exactly the reverse what we did with the automaton from Fig. 2 to get the simple automaton. In the parallel case, we add a new transition from the initial to the final state in the first automaton and call the second automaton at this transition.

The hierachical composition of simple attributed automata can be used to specify parsing of Dyck languages. In this case, the regular structure of the string is represented by transitions controlled by enabling predicates, balance of parentheses is accounted by at-

tributes. For every type of parentheses, one attributed automaton is used that calls another automaton when another type of a parentesis appears. In Fig. 4 the system of attributed automata for the recognition is shown for two types of parentheses used.

2.3. *Generalizations of Attributed Automata*

Transformational Attributed Automata

Because arbitrary attribute domains are allowed, the presence of input tape is not necessary. It can be modelled by the special component of attributes in the automaton without input tape as shown in (Meriste and Penjam, 1995; Meriste and Penjam, 1995b; Meriste and Vene, 1995).

DEFINITION 8. A *transformational attributed automaton* is a transition network $M = (S, T)$, where:

- S is a finite set of states with two distinguished states: an initial state $s_0 \in S$ and a final state $s_F \in S$. Every state $s \in S$ is associated with an attribute $a_s \in A_s$;
- $T \subseteq S \times S$ 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 $f_t : A_s \longrightarrow A_{s'}$.

Enabling predicates and transformation functions both have to be computable, but otherwise are arbitrary.

The functioning of the transformational automaton is analoguous to the case of finite attributed automata: it starts from the initial state s_0 with the initial attribute value $x = a_0 \in A_{s_0}$. A transition from one state two another is possible only if the corresponding enabling predicate is true. The transition is accompanied by evaluation of the attribute of the next state using the associated transformation function. The automaton stops if there is no transitions enabled. If this happens in the final state s_F, then the automaton finished successfully and the current value of the attribute $y = a_F \in A_{s_f}$ is treated as the output of automaton. This situation is denoted by $M(x) = y$. If automaton stops in some other state or doesn't stop at all, then the automaton fails and this is denoted by $M(x) = \bot$. This explains why such automata are called transformational – their only effect is the transformation of inputs into outputs.

In general, a transformational automaton specifies a relation between it's input and output attribute domains. If the automaton is deterministic (i.e., in every state there is at most one transition enabled), then the specified relation is a function. In (Meriste and Vene, 1995), it has been shown that even in the case of so called deterministic "primitive automata" – where all attribute domains are tuples of natural numbers, enabling predicates are checking the component of tuple to be zero and transformation functions are successor, predecessor or constant functions – every partially recursive function can be specified by it. It means that transformational attributed automata can be used as general model of (algorithmic) computation.

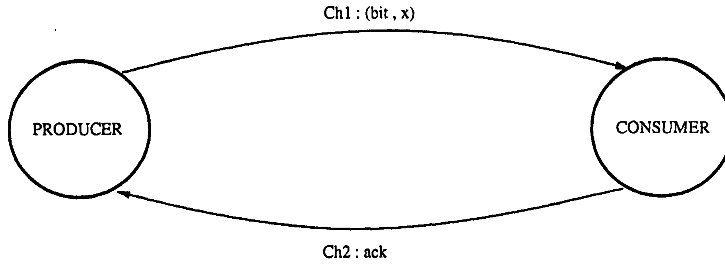


Fig. 5. A client-server system.

Interactive Attributed Automata
When one considers the modelling of interactive systems (Wegner, 1995), transformational attributed automata become inadequate. It's because of they have initial/final attributes as only means of communication with the outer world. In (Penjam, 1994) an extension of attributed automata with additional input/output primitives was considered and used to specify the Alternating Bit Protocol (ABP).

Consider the classical client-server network with two entities connected by two channels (see Fig. 5). The server (PRODUCER) sends messages through the channel $Ch1$ to the client (CONSUMER). Every message consists of two components – the data frame x and the header *bit* which under normal transition without errors should alternate 0 and 1 for successive frames. Through the channel $Ch2$ acknowledgements, which are also bits, can be passed.

In Fig. 6, the ABP is modelled using two separate attributed automata for PRODUCER and CONSUMER respectively. Operations $?Ch1(x, y)$ and $?Ch2(z)$ are representing reading messages from channels $Ch1$ and $Ch2$ respectively, and storing the values x, y and z into appropriate attributes. Operations $!Ch1(x, y)$ and $!Ch2(z)$ are representing sending the attribute values as messages to appropriate channels. Procedures $IN()$ and $OUT(x)$ "produce" and "consume" the data exchanged using the modelled mechanism.

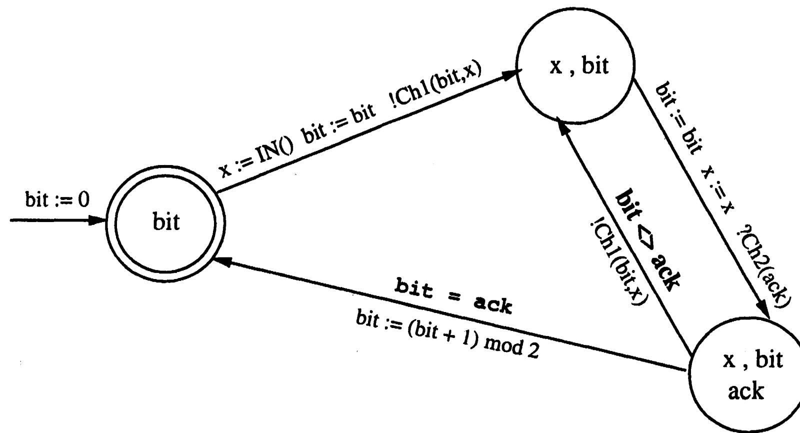
3. A Functional Specification of AA

3.1. *The Representation of Attributed Automata*

An attributed automaton M takes an initial attribute value $a \in A_{s_0}$ together with an input string $w \in I^*$ and returns the computed value together with the remaining input string (i.e., the pair $(a, w') \in A_{s_F} \times I^*$). In general, the automaton can be non-deterministic. It means that automaton is a relation between $A_{s_0} \times I^*$ and $A_{s_F} \times I^*$. Instead of this we treat automaton M as a set-valued function $A_{s_0} \times I^* \rightarrow \{A_{s_f} \times I^*\}$. The empty set as the result denotes failure; the singleton set means that only one recognition for the given string is possible.

By abstracting away from the concrete attribute domains of initial and final states we define the type of attributed automata as follows:

PRODUCER



CONSUMER

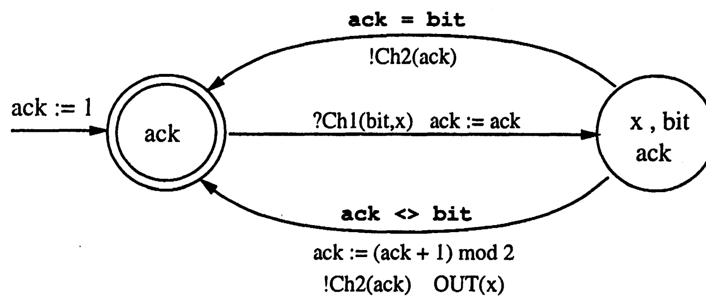


Fig. 6. An attributed model of the ABP.

$$\mathcal{AA} \doteq \forall a\, b.\, a \times I^* \longrightarrow \{b \times I^*\}.$$

3.2. *Primitive Attributed Automata*

- The first primitive automaton consumes the first symbol from input string if it matches the given symbol and fails otherwise:

$$`\,.\,` : \forall a.\, I \longrightarrow \mathcal{AA}(a, a),$$
$$`i` \doteq \lambda(a, w).\, \begin{cases} \{(a, w')\} & \text{if } w = iw', \\ \{\} & \text{otherwise.} \end{cases}$$

- The second primitive automaton checks whether the input string is empty or not:

$$\varepsilon : \forall a.\ \mathcal{AA}(a,a),$$
$$\varepsilon \doteq \lambda(a,w).\ \begin{cases} \{(a,w)\} & \text{if } w = \varepsilon, \\ \{\} & \text{otherwise.} \end{cases}$$

- The next primitive automaton corresponds to the predicate at the transition. It succeeds if the predicate holds and fails otherwise:

$$\cdot\,? : \forall a.\ (a \longrightarrow \mathbf{bool}) \longrightarrow \mathcal{AA}(a,a),$$
$$P\,? \doteq \lambda(a,w).\ \begin{cases} \{(a,w)\} & \text{if } P(a), \\ \{\} & \text{otherwise.} \end{cases}$$

- The fourth primitive automaton corresponds to the function at the transition. Using the function, it computes the new attribute value:

$$\cdot\,! : \forall a\,b.\ (a \longrightarrow b) \longrightarrow \mathcal{AA}(a,b)),$$
$$f\,! \doteq \lambda(a,w).\ \{(f(a),w)\}.$$

As the shorthand notation we denote the always failing automaton by $\Diamond$ and the identity automaton by $\Box$:

$$\Diamond \equiv (\lambda a.\mathbf{ff})\,? \qquad \Box \equiv id\,! \ \ (= \ (\lambda a.\ \mathbf{tt})\,?).$$

3.3. *Basic Composition Operators*

Next we need to define basic primitives for combining attributed automata – the sequential and parallel composition operators.

- In sequential composition, we apply the second automaton to the result of the first one. We have to take care of flattening the resulting set to make the composed automaton to have the correct type:

$$\star : \forall a\,b\,c.\ \mathcal{AA}(a,b) \times \mathcal{AA}(b,c) \longrightarrow \mathcal{AA}(a,c),$$
$$M_1 \star M_2 \doteq \lambda(a,w).\ \bigcup \{M_2(a',w') \mid (a',w') \in M_1(a,w)\}\,.$$

- In parallel composition, we apply both automata to the same input and take the union of the resulting sets as follows:

$$\oplus : \forall a\,b.\ \mathcal{AA}(a,b) \times \mathcal{AA}(a,b) \longrightarrow \mathcal{AA}(a,b),$$
$$M_1 \oplus M_2 \doteq \lambda(a,w).\ M_1(a,w) \cup M_2(a,w).$$

Another useful construction is the iteration of an attributed automaton zero or more times. It can be defined using the serial composition together with parellel composition as follows:

$$\cdot^* : \forall a.\ \mathcal{AA}(a,a) \longrightarrow \mathcal{AA}(a,a),$$
$$M^* \doteq (M \star M^*) \oplus \Box.$$

Also we use shorthand notation for the sequential composition of several '·' automata:

$$\text{“}w\text{“} \equiv \text{‘}i_1\text{‘} \star \text{‘}i_2\text{‘} \star \ldots \star \text{‘}i_n\text{‘} ,$$

where $w = i_1 i_2 \ldots i_n \ (n \geqslant 1)$.

EXAMPLE 1. The automaton from Fig. 1 can be defined as follows:

$$\begin{aligned} M_i &\doteq (\quad \text{‘1‘} \star (\lambda a.\ 2a+1)\,! \\ &\qquad \oplus\ \text{‘0‘} \star (\lambda a.\ 2a)\,! \quad)^*, \\ M_f &\doteq (\quad \text{‘1‘} \star (\lambda(a,b,c).\ (a, 2b+1, c+1))\,! \\ &\qquad \oplus\ \text{‘0‘} \star (\lambda(a,b,c).\ (a, 2b, c+1))\,! \quad)^*, \\ M_{b2d} &\doteq (\lambda a.\ 0)\,! \star M_i \star (\varepsilon \oplus \text{‘.‘} \star (\lambda a.\ (a,0,0))\,! \star M_f). \end{aligned}$$

The automaton M_{b2d} recognizes the binary number and converts it to the decimal form. First it initializes the attribute a and then uses the automaton M_i to recognize the integral part of the number. Then, if the whole input string is consumed it returns the decimal value of the number. Otherwise, if the remaining string starts with the decimal point, it uses the automaton M_f to recognize the fractional part.

The automaton recognizing the language $\mathcal{L} = \{a^n b^n c^n \mid n \geqslant 0\}$ can be defined as follows:

$$\begin{aligned} M \doteq\ & (\lambda x.\ 0)\,! && \star\ (\text{‘}a\text{‘} \star (\lambda k.\ k+1)\,!)^* \\ \star\ & (\lambda k.\ (k,0))\,! && \star\ (\text{‘}b\text{‘} \star (\lambda(k,l).\ (k,l+1))\,!)^* \\ \star\ & (\lambda(k,l).\ (k,l,0))\,! && \star\ (\text{‘}c\text{‘} \star (\lambda(k,l,m).\ (k,l,m+1))\,!)^* \\ \star\ & \varepsilon \star (\lambda(k,l,m).\ (k=l) \wedge (k=m))\,? \\ \star\ & (\lambda(k,l,m).\ k)\,! \end{aligned}$$

First three lines corresponds to the automata counting symbols a, b and c respectively. Then the automaton checks whether the input string is empty and all counters are equal. Finally, it returns the counter as its final attribute value.

4. Properties of Attributed Automata

Operators defined in the last section satisfy several nice algebraic properties. Here we list some of them which are most interesting.

- Sequential composition is associative with $\Diamond$ as the zero and $\Box$ as the identity element:

$$M_1 \star (M_2 \star M_3) = (M_1 \star M_2) \star M_3, \tag{1}$$

$$\Diamond \star M = \Diamond = M \star \Diamond, \tag{2}$$

$$\Box \star M = M = M \star \Box. \tag{3}$$

- Sequential composition of two primitive automata of a similar kind can be joined together:

$$\text{``}u\text{``} \star \text{``}w\text{``} = \text{``}uw\text{``}, \tag{4}$$
$$\varepsilon \star \varepsilon = \varepsilon, \tag{5}$$
$$P_1\,? \star P_2\,? = (P_1 \wedge P_2)\,? \tag{6}$$
$$f\,! \star g\,! = (f \circ g)\,! \tag{7}$$

- Parallel composition is associative, commutative and has $\diamond$ as its identity element:

$$M_1 \oplus (M_2 \oplus M_3) = (M_1 \oplus M_2) \oplus M_3, \tag{8}$$
$$M_1 \oplus M_2 = M_2 \oplus M_1, \tag{9}$$
$$\diamond \oplus M = M = M \oplus \diamond. \tag{10}$$

- Sequential composition distributes over parallel composition:

$$M_1 \star (M_2 \oplus M_3) = (M_1 \star M_2) \oplus (M_1 \star M_3), \tag{11}$$
$$(M_1 \oplus M_2) \star M_3 = (M_1 \star M_3) \oplus (M_2 \star M_3). \tag{12}$$

- Sequential composition of $\cdot\,?$ or $\cdot\,!$ with '$\cdot$' or ε is commutative:

$$P\,? \star \text{`}i\text{`} = \text{`}i\text{`} \star P\,? \tag{13}$$
$$P\,? \star \varepsilon = \varepsilon \star P\,? \tag{14}$$
$$f\,! \star \text{`}i\text{`} = \text{`}i\text{`} \star f\,! \tag{15}$$
$$f\,! \star \varepsilon = \varepsilon \star f\,! \tag{16}$$

The proof of the equational laws above is a simple calculation using the definitions of the corresponding operators together with the properties of set union. For instance the first law can be proven as follows:

$$\begin{aligned}
& M_1 \star (M_2 \star M_3) \\
= \; & \{\text{The definition of } \star\} \\
& \lambda(a,w).\bigcup\{\bigcup\{M_3(a'',w'') \mid (a'',w'') \in M_2(a',w')\} \mid (a',w') \in M_1(a,w)\} \\
= \; & \{\text{The associativity of } \textstyle\bigcup\} \\
& \lambda(a,w).\bigcup\{M_3(a'',w'') \mid (a'',w'') \in \bigcup\{M_2(a',w') \mid (a',w') \in M_1(a,w)\}\} \\
= \; & \{\text{The definition of } \star\} \\
& (M_1 \star M_2) \star M_3.
\end{aligned}$$

As a simple consequence of these laws is that our specification language is a proper generalization of the classical regular expressions:

COROLLARY 1. Expessions created using only $\diamond$, ‘ $\cdot$ ‘, $\star$, $\oplus$, $\cdot^*$ are regular exprssions.

It means that every (sub-)automaton created using only these five operations can be transformed to the equivalent minimal deterministic (sub-)automaton using well known algorithms from the classical automata theory.

Generally the automaton returns a set of results. If the automaton is deterministic, the result is always a singleton (if the automaton succeeds) or empty set (if it fails). Note that converse is not nessesarily true. The automaton M is called *unambiguous* iff for every possible input it either fails or returns singleton set. Otherwise the automaton is called *ambiguous*. Two automata *exclude each other* iff for every possible input the success of one yields the failure of the other.

Following easily provable facts can be used to determine whether an automaton is ambiguous or not:

- All primitive automata ‘ $\cdot$ ‘, ε, $\cdot$?, $\cdot$! are unambiguous.
- If automata M_1 and M_2 are unambiguous, then their sequential composition $M_1 \star M_2$ is also unambiguous.
- If automata M_1 and M_2 are unambiguous and exclude each other, then their parallel composition $M_1 \oplus M_2$ is unambiguous.
- If automata M_1 and M_2 are unambiguous and exclude each other, then the composition ${M_1}^* \star M_2$ is unambiguous.
- Primitive automata ‘i‘ and ε exclude each other.
- Primitive automata ‘i_1‘ and ‘i_2‘ exclude each other iff $i_1 \neq i_2$.
- Primitive automata P_1 ? and P_2 ? exclude each other iff $P_1 \supset \neg P_2$.
- If automata M_1 and M_2 exclude each other, then M_1 and $M_2 \star M_3$ also exclude each other.

Note also that sequential composition of two automata excluding each other is equivalent with always failing automaton $\diamond$.

5. Generalizations of Attributed Automata

5.1. *Transformational Attributed Automata*

A transformational attributed automaton can be defined as a set-valued function from the initial attribute value to the final one:

$$\mathcal{AA} \doteq \forall a\, b.\ a \longrightarrow \{b\}.$$

Because the type of attributed automata has changed we have to modify primitive automata and composition operators accordingly:

$$\cdot ? : \forall a.\ (a \longrightarrow \mathbf{bool}) \longrightarrow \mathcal{AA}(a,a),$$
$$P ? \doteq \lambda a. \begin{cases} \{a\} & \text{if } P(a), \\ \{\} & \text{otherwise.} \end{cases}$$

$$\cdot ! : \forall a\, b.\ (a \longrightarrow b) \longrightarrow \mathcal{AA}(a,b)),$$
$$f\,! \doteq \lambda a.\ \{f(a)\},$$

$$\star : \forall a\, b\, c.\ \mathcal{AA}(a,b) \times \mathcal{AA}(b,c) \longrightarrow \mathcal{AA}(a,c),$$
$$M_1 \star M_2 \doteq \lambda a.\ \bigcup \{M_2(a') \mid a' \in M_1(a)\},$$

$$\oplus : \forall a\, b.\ \mathcal{AA}(a,b) \times \mathcal{AA}(a,b) \longrightarrow \mathcal{AA}(a,b),$$
$$M_1 \oplus M_2 \doteq \lambda a.\ M_1(a) \cup M_2(a).$$

Definitions for $\Diamond$, $\Box$ and $\cdot^*$ stay the same. Also, it is easy to see that all relevant laws (not involving '$\cdot$' and ε) still hold.

5.2. *Interactive Attributed Automata*

Interactive attributed automata can be modelled using the tape(s) in the role of input-output channels. For instance to model the Alternating Bit Protocol from Figure 6 we define the type of attributed automaton as following:

$$\mathcal{AA} \doteq \forall a\, b.\ a \times Ch1 \times Ch2 \rightarrow \{b \times Ch1 \times Ch2\}.$$

Primitive automata $\cdot ?$, $\cdot !$ and composition operators $\star$, $\oplus$ should be redefined to follow the type of attributed automata. In addition, four new primitives should be defined for sending and receiving messages on both channels:

$$\begin{aligned}
\texttt{sendCh}_1 &: \forall a.\ \mathcal{AA}(a,a) \\
\texttt{sendCh}_1 &\doteq \lambda(a, ch_1, ch_2).\ \{(a, a : ch_1, ch_2)\} \\[1em]
\texttt{getCh}_1 &: \forall a\, b.\ \mathcal{AA}(a, a \times b) \\
\texttt{getCh}_1 &\doteq \lambda(a, b : ch_1, ch_2).\ \{((a,b), ch_1, ch_2)\} \\[1em]
\texttt{sendCh}_2 &: \forall a.\ \mathcal{AA}(a,a) \\
\texttt{sendCh}_2 &\doteq \lambda(a, ch_1, ch_2).\ \{(a, ch_1, a : ch_2)\} \\[1em]
\texttt{getCh}_2 &: \forall a\, b.\ \mathcal{AA}(a, b \times a) \\
\texttt{getCh}_2 &\doteq \lambda(a, ch_1, b : ch_2).\ \{((b,a), ch_1, ch_2)\}
\end{aligned}$$

Now, automata corresponding to the PRODUCER and Consumer can be defined as follows:

$$
\begin{aligned}
\text{PRODUCER} &\doteq (\lambda b.\, 0)\,!\, \star\, \text{PRO}_1 \\
\text{PRO}_1 &\doteq (\lambda b.\, (b, IN()))\,!\, \star\, \texttt{sendCh}_1 \star \text{PRO}_2 \\
\text{PRO}_2 &\doteq \texttt{getCh}_2 \star \text{PRO}_3 \\
\text{PRO}_3 &\doteq P_1\,?\, \star\, (\lambda(a,(b,x)).\, (b+1) \bmod 2)\,!\, \star\, \text{PRO}_1 \\
&\oplus P_2\,?\, \star\, (\lambda(a,(b,x)).\, (b,x))\,! \star \texttt{sendCh}_1 \star \text{PRO}_2
\end{aligned}
$$

$$
\begin{aligned}
\text{CONSUMER} &\doteq (\lambda a.\, 1)\,!\, \star\, \text{CON}_1 \\
\text{CON}_1 &\doteq \texttt{getCh}_1 \star \text{CON}_2 \\
\text{CON}_2 &\doteq (\;\; P_1\,? \star\, (\lambda(a,(b,x)).\, a)\,! \\
&\quad \oplus P_2\,? \star\, (\lambda(a,(b,x)).\, (a,(b, OUT(x))))\,! \\
&\quad\quad \star\, (\lambda(a,(b,x)).\, (a+1) \bmod 2)\,!) \\
&\quad \star\, \texttt{sendCh}_2 \star \text{CON}_1
\end{aligned}
$$

where $P_1(a,(b,x)) \iff a = b$ and $P_2 \doteq \neg P_1$.

6. Functional Parsers

In functional programming, recursive descent parsers are defined as functions from an input string into a list of parse tree / remaining string pairs:

$$\texttt{type}\ Parser\ a = String \longrightarrow [(a, String)]$$

together with some primitive parsers:

$$
\begin{aligned}
&zero && :: Parser\ a \\
&zero\ inp && = [\,] \\
\\
&result && :: a \rightarrow Parser\ a \\
&result\ a\ inp && = [(a, inp)] \\
\\
&item && :: Parser\ Char \\
&item\ [\,] && = [\,] \\
&item\ (c : inp) && = [(c, inp)]
\end{aligned}
$$

and composition operators:

$$
\begin{aligned}
&bind && :: Parser\ a \rightarrow (a \rightarrow Parser\ b) \rightarrow Parser\ b \\
&p\ `bind`\ f && = \backslash inp \rightarrow [f\ v\ inp' \mid (v, inp') \leftarrow p\ inp] \\
\\
&plus && :: Parser\ a \rightarrow Parser\ a \rightarrow Parser\ a \\
&p1\ `plus`\ p2 && = \backslash inp \rightarrow p1\ inp\ {+}{+}\ p2\ inp
\end{aligned}
$$

Here are some simple examples illustrating the use of basic parser combinators:

```
sat            :: (Char → Bool) → Parser Char
sat p          = item `bind` \x →
                   if p x then result x else zero

char           :: Char → Parser Char
char c         = sat (\y → x == y)

string         :: String → Parser String
string []      = [[]]
string (x : xs) = char x `bind` \_ →
                   string xs `bind` \_ →
                   return (x : xs)
```

The parser *sat p* consumes one character from input string. If the character satisfies the given predicate *p*, then the parser succeeds with the consumed character as the return value, otherwise the parser fails. It is used to define the parser *char c* which corresponds to the primitive attributed automaton '*c*'. Similarily the parser *string s* corresponds to the automaton "*s*".

One can see a great similarity between attributed automata and functional parsers. Indeed, the main difference between them is that while the former are parametrized by the initial attribute value, the latter are not. This makes the sequential composition of functional parsers a little bit more complicated and asymmetrical, as the transmission of computed values has to be explicit. The typical parser written in this style will look like:

```
M1 `bind` \a1 →
M2 `bind` \a2 →
      ⋮
Mn `bind` \an →
return (f a1 a2 ... an)
```

Note the role of λ-abstractions for explicitly binding temporary values. As a result, the distinction between control structure and attribute manipulation is not so clear as in the case of attributed automata.

Functional parsers are a little bit more powerful, as we can easily model sequential composition of attributed automata using '*bind*' as follows:

$$M_1 \star M_2 \doteq \backslash a_1 \rightarrow M_1(a_1)\ \text{`}bind\text{'}\ \backslash a_2 \rightarrow M_1(a_2)$$

But, the modelling of '*bind*' with terms of $\star$ is not possible.

As was noted in Section 4, sequential composition of attributed automata is associative and has □ as its identity; i.e., they form a monoid. In the case of functional parsers similar laws holds[2] for *bind* and *return*:

$$\begin{aligned}
p_1 \text{ `}bind\text{` } (\backslash a \rightarrow p_2 \text{ `}bind\text{` } f) &= (p_1 \text{ `}bind\text{` } \backslash a \rightarrow p_2) \text{ `}bind\text{` } f \\
return\ a \text{ `}bind\text{` } f &= f\ a \\
p \text{ `}bind\text{` } return &= p
\end{aligned}$$

In functional programming, the type constructor T, together with the operations $return :: a \rightarrow T\ a$ and $bind :: T\ a \rightarrow (a \rightarrow T\ b) \rightarrow T\ b$ which satisfy the laws given above, is called a *monad*. The notion is borrowed from cathegory theory where monads are used (among others) for modularizing the semantics of programming languages. In modern functional programming, monads are accepted as the basic tool to structure programs which deal with impure features like side effects, input-output, non-determinism, etc. We refer to (Wadler, 1992) for a good overview of how monads are used in functional programming. Exploiting the monadic structure of parsers gives an elegant way for their factorization and generalization. We refer to (Hutton and Meijer, 1996) for details.

7. Conclusion

We have developed a method of functional specification of attributed automata. It has good compositional and algebraic properties which allow systematic derivation of efficient implementations from readable specifications of attributed automata using correctness preserving transformations. Also, it is easily extendible to cope transformational, interactive and other different kind of attributed automata.

Our approach is very similar to the approach of defining recursive descent parsers in functional programming. Based on the simpler concept of monoids, instead of monads, it is less general. On the other hand, we achieve a cleaner separation between the regular control structure and attribute manipulations. It remains an open problem whether the loss of generality is outweighed by greater modularity or not.

References

Grönfors, T., and M. Meriste (1992). *Attributed Automata in Pattern Recognition of Digital Signals*, Res.Rep. R–92–1, Computer Science, University of Turku, Finland.

Hutton, G., and E. Meijer (1996). *Monadic Parser Combinators*.

Juhola, M., and M. Meriste (1992). An attributed automaton for recognising of nystagmus eye movements. In H.Bunke (Ed.), *Advances in Structural and Syntactic Pattern Recognition* , World Scientific, Singapore, pp. 194–203.

Kaljulaid, U., M. Meriste and J. Penjam (1993). *Algebraic Theory of Tape-Controlled Attributed Automata*. Res. Rep. CS59/93. Inst. of Cybernetics, Estonian Academy of Sciences, Estonia, Tallinn.

[2]Note, that the scope of the variable a, in the left hand side of the first equation (associativity) includes f, but excludes f in the right hand side. So the law holds only if the variable a does not occur free in f.

Meriste, M. (1994). *Attributed Automata – Some Results and Open Problems*. Res. Rep. CS75/94. Inst. of Cybernetics, Estonian Academy of Sciences, Estonia, Tallinn.
Meriste, M., and J. Penjam (1992a). Attributed finite automata. In: *Proc. of International Workshop on Compiler Compiler CC'92*, Technical report no 103, University of Paderborn, pp. 48–51.
Meriste, M., and J. Penjam (1992b). *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.
Meriste, M., and J. Penjam (1995a). *Attributed Models of Executable Specifications*, Res. Rep. CS80/95, Institute of Cybernetics, Estonian Academy of Sciences and Department of Computer Science, University of Tartu, Tallinn.
Meriste, M., and J. Penjam (1995b). Attributed models of computin, *Proc. Estonian Acad. Sci., Engineering*, 1, 2, 139–157.
Meriste, M., and V. Vene (1995). Attributed automata and language recognizers, In *Proc. of the Fourth Symposium on Programming Languages and Software Tools*. Visegrad, Hungary, 1995, pp. 114–121.
Penjam, J. (1994). *Attributed Automata: A Formal Model for Protocol Specification*. Res. Rep. CS71/93. Inst. of Cybernetics, Estonian Academy of Science, Estonia, Tallinn.
Wadler, P. (1992). Monads for functional programming. In M. Broy (Ed.), *Proc. Marktoberdorf Summer School on Programming Design Calculi*. Springer-Verlag.
Wegner, P. (1995). *ECOOP Tutorial Notes: Models and Paradigms of Interaction*, Tech. Rep. CS–95–21. Department of Computer Science, Brown University.

M. Meriste is born in 1950 in Estonia, graduated from Tartu University in 1972 in applied mathematics, Ph.D in computer science (1984). Currently the head of the Centre of Strategic Competence at Tartu University. His scientific interests include attribute methods in software construction, knowledge representation methods and systems, and telematics in higher education.

J. Penjam is born in 1955 in Estonia, graduated form Tartu University in 1979 as mathematician, Candidate of Science (Soviet equivalent of PhD) in computer science (1984). Currently professor and director of the Institute of Cybernetics at Tallinn Technical University. His scientific interests vary from semantics of programs and computational logic to artificial intelligence, constraint programming and high performance computing, networks.

V. Vene is born in 1968 in Estonia, graduated from Tartu University in 1994 in computer science. Currently Ph.D student in the Department of Computer Science at Tartu University. His scientific interests are programming language design and implementation, functional programming, type theory, semantic based program manipulation, category theory.

Atributinių automatų modeliai

Merik MERISTE, Jaan PENJAM, Varmo VENE

Atributinis automatas (AA) – tai formalizmas, skirtas specifikuoti koncepcines žinias, naudojant reguliarią sintaksę, papildytą atributais, aprašančiais konteksto sąlygotus ryšius bei semantines konceptų savybes. Atributinį automatą galima traktuoti kaip apibendrinimą baigtinio automato su atributais ir išskaičiuojamais ryšiais, jungiamais ir prie būsenų, ir prie automato perėjimų. Straipsnyje pasiūlytas naujas atributinių automatų specifikavimo metodas. Metodas sudarytas, panaudojus funkcinės kombinatorikos idėjas. Naudojant šį metodą, galima moduliarizuoti atributinių automatų specifikacijas. Metodas pasižymi geromis algebrinėmis savybėmis ir tinka specifikuoti įvairių rūšių atributinius automatus.