Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 985–990

DOI: 10.15439/2015F210
ACSIS, Vol. 5

Towards Machine Mind Evolution

Ján Kollár, Michal Sičák and Milan Spišiak
Department of Computers and Informatics,
Technical University of Košice
Letná 9, 042 00, Košice, Slovakia
Email: {jan.kollar, michal.sicak, milan.spisiak}@tuke.sk

***Abstract*—We introduce the principles of symbolization and conceptualization of perceived reality. We show them as processes that are applicable inside of a computer mind. We derive those processes from principles of a human mind. We present process of higher order regular abstraction and state automata acceptance as possible machine reality realization mechanism. We present the algorithm for evolution of machine mind language. Verification of this algorithm is presented in functional language Haskell. On the output, we have obtained fine-grained parallel non-redundant structure. It is in super-combinator form and represents elements of a machine mind language. Count of elements applications highly exceeds the count of elements themselves. Instead of accumulating acquired information, they are dissolved in the language of a mind. Vice versa, the information can be reconstructed from this language. We show that non-redundancy of a machine language is the decisive criterion for restructuring the machine mind language in each moment of communication. It is like that, so we can achieve more powerful and abstract communication between humans and computers or between computers themselves.**

I. Introduction

ABSTRACTION of the human-computer communication interfaces [1] can be increased by DSL development [2], coming out from the abstract syntax [3]. Grammatical and/or genetical language evolution [4], [5], [6] is more automated approach, but still oriented more to syntactic than to semantic facet of languages.

On the other hand, the development of thinking machine is a great challenge, which can be found in many works of historians, philosophers, psychologists and linguists. For example, Renfrew [7] characterizes humans as symbolic beings that language was evolved in a consequence of communication. Gardenfors [8] refers to the structured substance of concepts. Structured conceptualization for databases can be found in [9]. Chomsky's [10] hypothesis on the universal language of thought, innate to every human being is the subject of many discussions.

In this paper, we present the algorithm in which reality recognized by humans can be conceptualized and represented in the machine mind.

Our approach comes out from the empirical cognition of reality by humans and its reflection in the mind: our imaginations are structural, and our concepts that reflect the meaning of the reality are languages, since they are meaningful. In our opinion, no information acquired by human is being forgotten (it is just suppressed in our mind) and no information is stored multiplicity.

We deal with symbolization just marginally in this paper, more information can be found in [11]. We discuss conceptualization as a process of abstracting the language concepts in more detail, since differently abstracted concepts on input of the algorithm affect the evolved language of the machine mind.

We present the algorithm for the machine mind language evolution in eight subsequent steps. In this paper, we also introduce the principle of applicative deterministic automata, able to recognize reality incrementally.

Also another aspect of symbolized input processing is explored. We present possibility of higher order regular expressions creation.

Our results tend to distributed automata structures [12] and they conform with the principle of the determinism of systems [13]. The proposed solution allows us to separate process forming of language of the machine mind from the process of machine thinking reasoning on concepts. As we hope, this provides a perspective for cognitive and more abstract HC and CC communication.

II. Symbols and perception

Human is a symbolic creature, i.e. the human communication is symbol based. He creates immaterial concepts inside of his mind that flow right from the symbols that have material substance. Concepts exist only inside of a human mind, i.e. his imagination. Symbols posses their informal meaning and at the same time they might be structured with high degree of complexity during the communication.

Symbolization is a symbol based reality representation process. Simple components of reality, such as the letter A or an image of a square can be symbolized with simple symbols like a or b. Therefore we are able to symbolize perceived reality, like the sequence (1), into the symbol sequence $aabababa$.

$$AA\square A\square A\square A\square A \tag{1}$$

The new structured symbol has been created. The relation between the reality and its symbolized form is bidirectional. It is not enough to symbolize reality at the input in order to communicate, as the reality needs to be reconstructible at the output from the previously obtained symbol structure.

This work was supported by project VEGA 1/0341/13 Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication

The symbolization of the sequence (1) is expressed with the rule (2):

$$\frac{A \leftrightarrow a \qquad \square \leftrightarrow b}{AA\square A\square A\square A\square A \leftrightarrow aababababa} \tag{2}$$

Section above line in the rule (2) tells us, that if we have already identified reality of simple facts A and $\square$ and symbolized them with symbols a and b, then new structured reality (1) is going to be symbolized with the use of a new symbol. This is the structured symbol $aababababa$. The structure doesn't necessarily have to be a sequence as it is in our example, but may be any structure. Symbols are just abstractions of reality in our view, recorded and produced from inside of a memory during communication. In order for communication to be faultless, such records of reality are stored inside the memory. They represent reality inside a human or computer mind. Since a human does not think in symbols, we may need to imagine its relation to machine mind abstract symbols as a connective bidirectional links to records of reality. Since we do not posses an architecture of a machine that corresponds to human mind, we find no other way than to consider them as alphabet symbols - terminal symbols of a language, and to represent them in discrete way, like whole non-negative numbers.

The purpose of symbolization is not the same as recognition problem. A machine is able to register and reproduce reality. Cognition is tied with the language evolution and it is inherent property of a mind. The language evolution works together with symbol abstraction. It is based on conceptualization, whose substance is the concept creation inside of an internal machine language.

III. Conceptualization

A mechanism of conceptualization is a higher form of language abstraction than symbolization is.

String (3) is a concept - language, identical with the symbol sequence perceived on input. An abstraction was not performed upon it.

$$aababababa \tag{3}$$

Sequence (1) is a concept or a language that has meaning if we consider process of symbolization defined in (2) being applied.

Consider a concept in a form of a regular expression (4), where $(r)^*$ is transitive closure of an expression r. This concept is created by performed abstraction that searches for repeating patterns, in our case the sequence ab.

$$a(ab)^*a \tag{4}$$

This concept is more abstract than concept (3) for there is a relation (5) between the languages (3) and (4).

$$aababababa \subset a(ab)^*a \tag{5}$$

A machine operating with the language (3) cannot effectively communicate with a machine that recognizes the language (4), since its language is less abstract. In automaton theory it means that an automaton derived from the regular expression (3) accepts only terminal symbol string $aababababa$. It can be generated from the regular expression (4) however. On the other hand, the human or the machine (3) can learn from its counterpart (4), since it is capable of producing the symbol set aa, $aaba$, $aababa$, etc. It can achieve level (4) by this process. We can say, that language (3) will mutate to the language (4). The premise for machine learning is a massive stream of structured symbols at the input. Language (3) mutation into more abstract form is based on such a stream.

A sequence of symbols at the input shown at (6) leads to conceptualization - the process of concepts creation shown in (7).

$$aa; aaba; aababababa \tag{6}$$

$$\begin{aligned} &1.\quad (\{\}, aa) \Rightarrow \{aa\} \\ &2.\quad (\{aa\}, aaba) \Rightarrow \{a(ab)^{\{0,1\}}a\} \\ &3.\quad (\{a(ab)^{\{0,1\}}a\}, aababababa) \Rightarrow \{a(ab)^{\{0,1,3\}}a\} \end{aligned} \tag{7}$$

In the first step, the mind is enriched by language aa as there is the symbol aa at the input and the mind is empty. This language is stored inside of the machine mind. Next, the symbol $aaba$ arrives at the input. The result of conceptualization is mutation of the language aa inside the mind of a machine into the language $a(ab)^{\{0,1\}}a$ during the second step. Another symbol transforms the language into $a(ab)^{\{0,1,3\}}a$ during the third step. This language enables the machine to communicate in form of the symbols aa, $aaba$ and $aababababa$, since (8) holds.

$$\begin{aligned} &a(ab)^0a \Rightarrow aa \\ &a(ab)^1a \Rightarrow aaba \\ &a(ab)^3a \Rightarrow aababababa \end{aligned} \tag{8}$$

The conceptualization example (7) shows us, that symbols on the right side of (8) can be generated and then represented as records of reality from concepts on the left side. Those concepts are languages, since they are sub-languages of most general language (9). This language is a concept only when all symbols generated from it have meaning i.e. represent the reality.

$$a(ab)^{\{0,1\}}a \subset a(ab)^*a \tag{9}$$

Following facts about conceptualization are to be noted:

1) Conceptualization leads to language ambiguity, it expands exponentially, therefore a selection condition is necessary. For example, language $(a)^{\{2\}}$ could have been created in the first step of (7) instead of the language aa.
2) It is possible to represent single symbol with multiple languages. For example, if language $a(ab)^{\{0,1,3\}}a$ is created inside of a mind of one machine and language $aa(ba)^{\{0,1,3\}}$ inside of another, they will be equal in terms of inter-machine communication. Machines will "understand" each other.

We can conclude that the machine mind is non-redundant i.e. does not contain any two identical language components. The mind hasn't got an ability to forget and it is deterministic. Language structure of the mind has to be highly parallel, associative, and amount of interconnections between language components highly exceeds the amount of components themselves.

IV. Structured symbols as regular expressions

We have shown, that reality can be symbolized as a structured symbol describable by a regular grammar. In this paper, we use regular expressions to describe concepts that were obtained by previously symbolized reality. Those expressions themselves provide interesting ways to describe concepts, as we will show in this section.

Regular expressions are usually an in-line way to describe regular grammars. In practice, we use them for string matching. Therefore, fundamental data type (the type of symbol that regular grammar itself is matching to patterns) for accepting is a character. Our expressions are composed of operators sequence, closure and sum. First two have been already described earlier. Sum operator is n-ary and designated as | symbol.

Regular expressions represented as a string itself are not well executable. One of better executable solutions is to use a tree form that can be considered meta-executable, as described in [4]. All our experiments were performed in functional language Haskell. It allows easy creation of complex data structures, such as a tree. Definition is based on work [11] and is depicted below:

```
data RExp a = TERM a | SEQ [RExp a] |
    SUM [RExp a] | CLS (RExp a)
```

Data structure `RExp` contains four constructors. Constructor `TERM` represents fundamental type, `SEQ` is for the sequence, `SUM` represents the sum operation and `CLS` is for the closure. It is notable, that we are not using type `Char` as the fundamental type for regular expression. Current definition allows us to match expressions composed of an arbitrary data type. We will return to this proposition later on. Sum and sequence operators are n-ary.

Consider following regular expression:

$$a(b|c)^* \tag{10}$$

where $a, b, c \in \Sigma$, where Σ is terminal symbol set. This expression matches string that begins with symbol a and after that an arbitrary long string consisting of bs and cs is matched. Transformed regular expression into the tree form is shown bellow. Transformation is based on scheme described in [11], so we won't present it here for brevity.

```
SEQ [TERM 'a',
     CLS (SUM [TERM 'b',TERM 'c'])]
```

As mentioned earlier, fundamental type of regular expression does not have to be of type `Char`. Definition allows us to use arbitrary data type.

TABLE I
Meaning of regular expression elements

symbol	meaning
a	$d(e)^*$
b	ef
c	$(d\|f)$

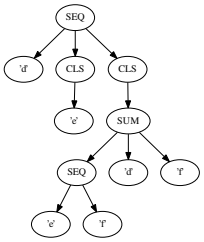

Fig. 1. Less abstract regular expression

Let us consider our regular expression data type as a form of abstract interpretation of another regular expression. Therefore elements a, b and c represent other regular expressions, and the resulting type will be `RExp (RExp Char)`. This can be considered a higher order regular expression.

Term "higher order" is used similarly as in functions, where higher order functions take a function as a parameter or return one. Since our regular expression takes another as a parameter, it can be considered higher order regular expression.

Definitions of symbols the a, b and c meaning is depicted in Table I. We use different symbols, d, e and f, in this abstraction to make it clear on what level of we currently are. By substitution of the elements a, b and c we get less abstract regular expression:

$$d(e)^*(ef|(d|f))^* \tag{11}$$

Tree form of expression (11) is shown in Fig 1.

Fundamental data type can be arbitrary, even recursive. Question arises, what if we would be able to create data type that is going to be recursive and will contain `RExp a` type. Data type like that would need a terminating and recursive part. In our case, we will use type shown bellow. Regular expression itself would be of type `RExp RexChar`.

```
data RexChar =
      Ch Char | R (RExp RexChar)
```

Every element of regular expression can be either a character (the use of `Char` is not obligatory, we have chosen this type for example purposes) or regular expression of same type as the original expression. Purpose of recursive regular expressions is that they can contain themselves as their fundamental elements entering form of recursion.

In Haskell we can create those expressions statically as a source code or we might obtain them dynamically during execution by parsing input. Here is an example of a higher order regular expression in static Haskell code:

```
a,b :: RExp RexChar
```

```
a = SEQ [TERM (Ch 'a'),
         SUM [TERM (R b), TERM (Ch 'c')]]
b = SUM [SEQ [TERM (Ch 'b'),
         TERM (R b)], TERM (CH 'b')]
```

We can rewrite them as:

$$\begin{aligned} A &\rightarrow a(B \mid c) \\ B &\rightarrow bB \mid b \end{aligned} \tag{12}$$

If we consider the rules of CFG from Chomsky hierarchy, we can see that our recursive regular expressions (12) are written in format of a BNF rule. We can see, that our recursive regular expressions are corresponding with CFG rules and BNF grammars.

By definition of hierarchy of languages, for every regular language a finite state automaton can be constructed. Automata are efficient way for accepting or even generating phrases of regular languages. Next section discuses the idea of higher order state automata, that may not necessarily be finite, yet for all practical purposes, they will be usable.

V. State automata derived from higher order expressions

As machine mind processes symbolized input and transforms it into concepts, it needs mechanism for structured symbols acceptance. This can be done with state automata. Deterministic automata can only have one transition for each symbol from one state to another and cannot contain empty transitions. Every nondeterministic automaton can be converted into deterministic. For practical purposes, it is always better to have control over choosing which step to take next.

To obtain state automata we need to have an expression first. We will use following example:

$$A \rightarrow aAb|ab \tag{13}$$

Expression (13) can be interpreted as a grammar of context free language that contains strings that have exact number of as and bs in succession. This is typical illustration of a non-regular context free language, uninterpretable as a regular expression. However with our recursive expression it is possible. Resulting tree created from (13) will be infinite, depicted on Fig 2. We can see that second argument of sum operator, ab, is terminator part. We can conclude that if we consider our regular trees as executable, they are able to accept a context free language.

In work [11] is presented an algorithm that transforms regular expressions directly into minimum state automaton. Same can be applied to higher order regular expressions. Resulting automaton of expression (13) is depicted in Fig 3. This is an automaton corresponding to regular expression labeled as A.

There is a transition within body of an automaton in Fig 3 that accepts not a symbol but entire automaton. In this case it accepts itself. We can perceive this fact as a recursive jump to the higher level. For an input string "$aaabbb$", our automaton would have jumped twice into higher levels. We could see this

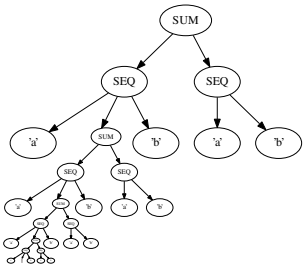

Fig. 2. Tree form of expression (13).

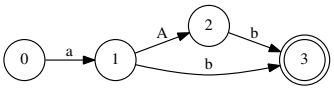

Fig. 3. Context free automaton

as an automaton with infinite levels, as depicted in Fig 4. We see, that those automata are equivalent to recursive transition networks.

Final state on higher level does not always mean possible string acceptance, it can just represent return from a recursion, as depicted in Fig 4, where states 3 of all higher levels have transition into state 2 of lower level. Acceptance of string is achieved only in final states on base level. This does not mean, that higher level final states are not allowed to have transitions into final states of lower levels. In case like that, whole string of symbols can be accepted at higher level and then sequence of empty transitions into base final state will occur.

Consider following higher order expression written according to our rules, labeled as B:

$$B \rightarrow ba(Bb|(b)^1) \tag{14}$$

Closure in last symbol is parametrized and it designates zero or one repetition. Automaton for expression (14) is depicted in Fig 5. In this particular case see an example of higher order nondeterminism.

Automaton itself is deterministic, if we look at it as residing on one level, where B transition is just another symbol transition. It is unfortunately not our case and B means

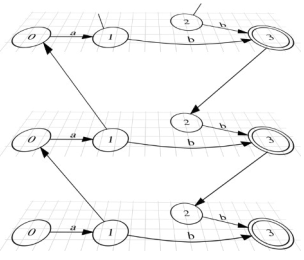

Fig. 4. Recursive infinite state automaton

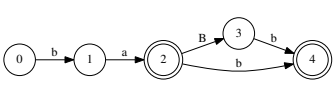

Fig. 5. Nondeterministic higher order state automaton

recursion, where automaton accepts itself. Consider execution where we are on non-zero level and are in the state 2. It is the final state, so execution on current level can terminate and we can return back. There is also a transition into the higher level and a normal transition with the symbol b.

Therefore exactly three transitions are possible at this moment. And all three are accepting one symbol, b. Jumping into the higher level causes jump into state 0 and here only b is accepted. Returning one level down will get us into the state 3 and here b is only acceptable symbol as well.

Application of automata inside machine mind is a possible solution for perceived reality acceptance, and as we have shown, symbolized structures do not need necessary be simply structured. The accepting part is but a small portion of a whole concept of the machine mind. The more important part is to deal with symbols and conceptualize them into the memory. We will continue to use only simple regular expressions further on for the sake of simplicity. Our focus is to get closer to human comprehension of concepts as possible. In the next section we will present the algorithm of abstract machine mind evolution that satisfies conditions of symbolization and conceptualization laid in previous sections.

VI. Abstract language evolution algorithm

Let the language $a(ab)^*a$ be a result of conceptualization, that is also a concept, since symbolization process was bound to reality. This concept is composed of infinite amount of sub-concepts, where each of them has its own meaning. The most simple concept is aa that symbolizes meaning of two uppercase letters AA. We can abstract from language semantics, since it is defined by separate process of symbolization. Therefore we can see them as simple regular expressions rather than complex languages.

An algorithm for abstract language evolution is composed of those eight steps:

1) Transformation of regular expression to metaexecutable form of (meta)syntactic tree, considering priority and associativity of metaoperations.
2) Selection of arguments terminal symbols and transformation of each (sub)expression to the form of its application to selected arguments.
3) Abstraction of each expression deriving the arity of lambda abstraction from number of arguments, selected in step 2. Elementary languages arise being still applied in the tree.
4) Decomposition of elementary languages (elementary language concepts). Elementary languages are dissolved in associative memory represented by a list and applied by the MATCH operation.
5) Abstraction of all applications (APP) to terminal symbols, i.e. replacing them by abstract applications (APPA) to lambda variables. The set of mutually applied abstract languages is formed.
6) Compression - removal of multiple occurrences of the same abstract languages, i.e. removal of redundancy of elementary languages.
7) Filtration - removal of sets of pointers to redundant languages. Manipulation phase.
8) The machine mind language extraction.

The sequential separation of steps is useful for the methodological purposes and also for verification of evolution on a sequential computer. But it can be noticed that sequential steps are executable potentially in parallel as pipelined processes.

VII. Algorithm implementation using Haskell

In the Haskell implementation, meta-operations of sequence and transitive closure used in abstract concept $a(ab)^*a$ as well as sum (|) meta-operation were used. This allows us to apply the algorithm to the abstract language concept $a(b)^*|c$ as an input sample.

Each elementary language is in supercombinator form, i.e. lambda abstraction (LAM n e), where n represents lambda variables $x_1, \ldots, x_n$ and e is a meta-expression. Meta-expression e in the form VAR 1 represents lambda variable x_1, otherwise it is the application of binary sequence metaoperation (SEQ), unary transitive closure metaoperation (CLS), and n-nary sum metaoperation (SUM), all being applied to the applications of elementary languages. These applications (APPA (MATCH k) $[x_1, \ldots, x_m]$) apply k-th elementary language of arity m to bound lambda variables. By other words, k-th elementary language is a supercombinator, which can be accessed to be applied in a highly distributed associative memory of mind by operation (MATCH k) .

VIII. Algorithm verification

We obtain the abstract language concept $a(b)^*|c$ represented in the machine mind by five elementary languages L in supercombinator form (15).

$$\begin{aligned} L &= \{L_0, L_1, L_2, L_3, L_4\} \\ L_0 &= \lambda x_1.x_1 \\ L_1 &= \lambda x_1.L_0\ x_1 \\ L_2 &= \lambda x_1.(L_0\ x_1)^* \\ L_3 &= \lambda x_1.\lambda x_2.L_1\ x_1 + L_2\ x_2 \\ L_4 &= \lambda x_1.\lambda x_2.\lambda x_3.(L_3\ x_1\ x_2|L_0\ x_3) \end{aligned} \tag{15}$$

Each of elementary languages is applied to admissible abstract symbols, that sets have been derived in the algorithmic evolution. All possible applications (16) represent all meaningful sub-concepts acquired by the most complex input concept.

$$\begin{aligned} &L_0\ a,\ L_0\ b,\ L_0\ c \\ &L_1\ a \\ &L_2\ b \\ &L_3\ a\ b \\ &L_4\ a\ b\ c \end{aligned} \tag{16}$$

For example, we can verify that the applications of identity supercombinator L_0 yields concepts a, b, and c, that represent facts, and the application of supercombinator L_4, see (17), yields the most complex concept $a(b)^*|c$.

$$L_4\ a\ b\ c = a(b)^*|c \tag{17}$$

Abstraction in the process of conceptualization significantly decreases the amount of elementary supercombinators of the machine mind. For example, abstract concept $a(ab)^*a$, i.e. $a(ab)^ka$ would be represented by 6 supercombinators, for each $k \geq 0$. But if the algorithm is applied to non-abstracted concept (such as $aabababa$), the amount of supercombinators grows linearly ($2k + 3$ for $k \geq 0$, i.e. 9 supercombinators for $aabababa$).

Summarization of structurally identical concepts yields no change of supercombinator form of languages L, just that amount of their arguments is increased. For example, (18) holds.

$$L(a(ab)^*a|c(cd)^*c) = L(a(ab)^*a|c(cd)^*c|e(ef)^*e) \quad (18)$$

Considering well abstracted conceptualization of large amount of symbols on input, we can await that the amount of application bindings exceeds the amount of elementary languages rapidly, preserving non-redundancy of the set of elementary languages.

IX. Recognition by applicative automata

As can be seen, derived elementary languages are defined by regular expressions in that a single meta-operation is applied to elementary languages applications, instead of terminal symbols. Let us remind that:

$$L_4\ a\ b\ c = a(ab)^*c$$

In the first step of meta-computation we get:

$$L_4\ a\ b\ c \Rightarrow L_3\ a\ b\,|\,L_0\ c = A\,|\,B$$

Then it is not necessary to recognize concrete sub-concept belonging to $a(ab)^*a$ by generation of DFA for $a(ab)^*a$, but it is sufficient to derive the applicative DFA for regular expression $A|B$, where terminal symbol A represents the set of final states of applicative DFA for ($L_3\ a\ b$) and terminal symbol B represents the set of final states of applicative DFA for application ($L_0\ c$).

Considering (15), the number of applications commonly highly exceeds the number of elementary languages. That is why we can formulate the hypothesis on saturation of the machine mind, which can be realized as slowing-down expansion of set the L with each new concept on input. Multiple occurrences of terminal symbols in arguments in (16) are of less importance, since they physically do not exist, because they represent just interconnections to reality records.

X. Conclusion

The main advantage of internal machines supercombinator form is addition of automatic conceptualization criterion. It is the minimal count of elementary languages, which themselves are supercombinators as well. Internal language of a mind does not expand with ongoing stream of symbols on input. It is common knowledge that almost everything is new for a little child but an adult hardly finds something new that he hasn't heard or seen. In order for learnable machine to communicate with human, it needs to work on similar thought principles. This thinking is language oriented. If this principle is present, it can then absorb reality, symbolize it and based on abstraction it can create concepts in form of internal languages of a mind.

Our position isn't that supercombinator form is the only form able to represent internal language of mind. It is but an algorithmic realization of a possible way not to store anything multiplicity inside mind but to remember everything. Our supercombinator form has future applications possibilities despite the fact that each of cognition steps could look differently, if someone else performed it. Experiment with higher order expressions shows us that our future work may be involved in other forms of grammars.

We do not think the machine mind evolution is based just on regular expressions, as presented in this paper. But using this simplest case we were able to introduce the principle of applicative automata for output communication. Clearly, machines should be active in communication, i.e. able to formulate the questions.

Derived language of the machine mind consists of elementary languages, such that each of them can be fetched associatively by matching. An interesting structural similarity between the machine mind and neural brain network can be noticed. Both mind meta-operations and brain neurons are the nodes and both mind applications and brain synapses are arcs of a graph. However, in our opinion, no bidirectional relation between brain and mind can be deduced from mentioned structural similarity.

References

[1] M. Bačíková, J. Porubän, and D. Lakatoš, "Defining domain language of graphical user interfaces." in *SLATE*, 2013, pp. 187–202.

[2] J. Poruban, M. Bacikova, S. Chodarev, and M. Nosal, "Pragmatic model-driven software development from the viewpoint of a programmer: Teaching experience," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*. IEEE, 2014, pp. 1647–1656.

[3] J. Porubän, M. Forgáč, M. Sabo, and M. Běhálek, "Annotation based parser generator," *Computer Science and Information Systems*, vol. 7, no. 2, pp. 291–307, 2010.

[4] J. Kollár and E. Pietriková, "Genetic evolution of programs," *Central European Journal of Computer Science*, vol. 4, no. 3, pp. 160–170, 2014.

[5] F. Javed, M. Mernik, B. R. Bryant, and A. Sprague, "An unsupervised incremental learning algorithm for domain-specific language development," *Applied Artificial Intelligence*, vol. 22, no. 7-8, pp. 707–729, 2008.

[6] M. O'neill, C. Ryan, M. Keijzer, and M. Cattolico, "Crossover in grammatical evolution," *Genetic programming and evolvable machines*, vol. 4, no. 1, pp. 67–93, 2003.

[7] C. Renfrew, *Prehistory: the making of the human mind*. Random House Digital, Inc., 2009, vol. 30.

[8] P. Gärdenfors, "Symbolic, conceptual and subconceptual representations," in *Human and Machine Perception*. Springer, 1997, pp. 255–270.

[9] M. Pater and D. E. Popescu, "Multi-level database mining using afopt data structure and adaptive support constrains." *International Journal of Computers, Communications & Control*, vol. 3, no. 3, 2008.

[10] N. Chomsky, *Syntactic structures*. Walter de Gruyter, 2002.

[11] J. Kollár, "Formal processing of informal meaning by abstract interpretation," *Smart Digital Futures 2014*, vol. 262, p. 122, 2014.

[12] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 207–218.

[13] J. L. Peterson, *Petri net theory and the modeling of systems*. Prentice-hall Englewood Cliffs (NJ), 1981, vol. 132.