# On Parameter-Driven Generation of Algorithm Schemes

Olena Yatsenko

Institute of Software Systems of National Academy of Sciences of Ukraine,
Glushkov prosp. 40, 03187 Kyiv, Ukraine
oayat@ukr.net

**Abstract.** The approach to development of serial and parallel algorithms, which is based on usage of parameter-driven generation of algorithm schemes, is proposed. The approach uses algebras of algorithms and hyperschemes, and also structural design grammars. Hyperschemes are parameterized specifications that allow to receive the algorithms adapted to specific conditions of usage. The paper also describes the developed software tools for designing and synthesis of algorithms and programs.

**Keywords:** hyperscheme, regular scheme, structural design grammar, synthesis of programs, systems of algorithmic algebras.

## 1 Introduction

Algebraic algorithmics (AA) is one of significant directions of computer science, which has arisen on a joint usage of algebra, logic and algorithm schemes within the framework of Ukrainian algebraic-cybernetic school [1, 4, 5, 8]. It formalizes the knowledge about subject domains with the help of algebraic facilities and deals with problems of formalization, substantiation of correctness and transformation of algorithms. AA uses high-level specifications of programs, represented in Systems of Algorithmic Algebras (SAA) [4, 5]. One of the essential problems of AA is to increase the adaptability of programs to specific conditions of their use. In particular, the problem can be solved at the expense of usage of parameter-driven generation of algorithm specifications by means of higher level algorithms, which are called hyperschemes [9].

In this paper the approach to generation of serial and parallel algorithms on the basis of algebras of hyperschemes is proposed. Hyperschemes are parameterized algorithms for solving a certain class of problems; setting specific values of parameters and subsequent interpretation of a hyperscheme allows to receive algorithms adapted to specific conditions of their use. Hyperschemes are adjacent to well-known methods of transformational synthesis: term rewriting systems [3, 7], mixed computations [2], macrogeneration [6]. The proposed approach is based on algebraic-grammatical models, developed in work [9]. The novelty of this paper consists in application of hyperschemes to representation of derivation algorithms in structural design grammars [1], which are sets of rules for generation of algorithm schemes, specified in SAA. Furthermore, this work includes the development of a

software tool for construction and interpretation of hyperschemes. The mentioned tool is one of basic components of Integrated toolkit for Designing and Synthesis of programs (IDS) [5]. IDS uses algebraic specifications of algorithms and applies three interdependent forms of knowledge representation: analytical (formulae), natural language text and visual (flow charts) [5]. The main feature of IDS consists in usage of the method of interactive design of syntactically correct algorithms [1], which is oriented to elimination of syntax errors during construction of algorithm schemes.

The outline of the paper is the following. Section 2 is devoted to a concept of algebra of hypershemes and associated processes of algorithm generation. Section 3 considers structural design grammars and inference mechanisms, based on hyperschemes. Section 4 describes software tools, developed for automation of a process of algorithm generation. The proposed approach is illustrated on sorting and linear algebra algorithms.

## 2 Algebra of Hyperschemes and Generation of Algorithms

In this section the algebra of hyperschemes (AHS) is considered, which is the formalism that is used for parameter-driven generation of algorithms, specified in SAA. Definition of AHS is similar to SAA, so SAA is defined first. System of Algorithmic Algebras is a two-sorted algebra

$$SAA = < \{U, V\}; \Omega >, \tag{1}$$

where $U$ is a set of logical conditions (predicates) and $V$ is a set of operators, defined on an information set $P$. $P$ is a set of all data (input, output and intermediate), being processed by algorithms. Each predicate from set $U$ is a function that maps elements of $P$ to elements of a set $\{0, 1, \mu\}$, where 0 is for false, 1 is for true and $\mu$ is for unknown. The value $\mu$ is used to indicate that an error has occurred during the computation of a condition [9]. The unknown state $\mu$ can be metaphorically thought of as a sealed box containing either an unambiguously true or unambiguously false value. The knowledge of whether any particular $\mu$ state secretly represents true or false at any moment in time is not available. Operators from set $V$ map information set to itself. $\Omega = \Omega_1 \cup \Omega_2$ is the signature of operations consisting of a system $\Omega_1$ of logical operations and a system $\Omega_2$ of operations of serial and parallel execution of operators. The mentioned operations will be considered below.

Operator representations of algorithms in SAA are called regular schemes (RS). The algorithmic language SAA/1 [1, 4, 5] is based on mentioned algebra and is used to describe algorithms in a natural language form. The algorithms, represented in SAA/1, are called SAA schemes.

Operators and predicates can be basic or compound. The basic operator (predicate) is the operator (predicate), which is considered in SAA schemes as primary atomic abstraction. The compound predicates are constructed from basic ones by logical SAA operations [4]:

- disjunction: $\alpha \vee \beta$;
- conjunction: $\alpha \wedge \beta$;

• negation: $\overline{\alpha}$ .

Compound operators are built from elementary ones by means of operations of serial and parallel execution operators:

• composition $A * B$, which is the operation of serial execution of operators $A$ and $B$ (this operation can also be written as $A$ THEN $B$);

• $([\alpha] A, B)$ is the conditional operation that executes operator $A$ if condition $\alpha$ is true, and applies operator $B$ otherwise (further this operation are called conditional operation);

• $\{[\alpha] A\}$ is a while loop, which executes operator $A$ repeatedly until condition $\alpha$ is true;

• $SELECT(\alpha_1 \to A_1, \alpha_2 \to A_2, \ldots, \alpha_n \to A_n)$ is the switch operation, that executes the first operator among operators $A_i$, if the value of corresponding condition $\alpha_i$ is true and then breaks execution without checking the values of other conditions $\alpha_i$;

• operation $A // B$ executes two operators in parallel;

• control point $CP(\alpha)$ is used to synchronize parallel processes. Each control point is assigned with a condition $\alpha$, which is false until the computation process reaches the point and becomes true at the moment of reaching the point;

• synchronizer $S(\alpha)$ is also used in parallel algorithms; it delays the computation until the value of condition $\alpha$ is true (this value is generally changed by some control point).

SAA is a formal tool that can be used for solving problems of theoretical programming, such as optimization and verification of algorithms, formalization of semantics of programming languages etc. [1, 4, 5, 9].

Algebra of hyperschemes is also a two-sorted algebra

$$AHS = < \{U', V'\}; \Omega' >, \tag{2}$$

where $U'$ is a set of logical conditions (predicates) and $V'$ is a set of operators; $\Omega'$ is a signature of operations. The set of conditions is associated with parameters, which control the process of generation of an algorithm. The operations from signature $\Omega'$ are similar to SAA. The difference from SAA is that predicates from set $U'$ map elements of information set $P$ to elements of a set of $\{0, 1, \mu, \eta\}$, where additional value $\eta$ stands for “not computed”. The element $\eta$ is used to indicate that the value of a condition cannot be calculated due to the lack of information about parameter values [9]. The instance of such condition is considered further in Example 1. The truth tables for this 4-valued logic are given in [9].

Execution of operator $A \in V'$ at a current state $p \in P$, leads to a transition to a new state $A(p) \in P$ and generation of a fragment $F(A, p)$ of an output scheme of an algorithm. $F(A, p)$ is the function that specifies the generation technique for all operations of AHS and will be considered below.

By analogy with SAA, operator representations of algorithms in AHS are called regular hyperschemes (RHS). Each RHS $A$, being applied to a state $p \in P$, generates RS $F(A, p)$.

The function $F(A, p)$ for the main SAA operations was defined in paper [9]. Particularly, the composition operation $A * B$ generates the operator $C = A * B$ without changes, according to the function $F(C, p) = F(A, p) * F(B, p)$, where $p \in P$.

The conditional operation ($[\alpha] A, B$) (see the definitions of SAA operations given above) generates the operator $C = ([\alpha] A, B)$ such, that for each $p \in P$

$$F(C, p) = \begin{cases} F(A, p), \text{if } \alpha(p) = 1; \\ F(B, p), \text{if } \alpha(p) = 0; \\ ([\alpha[p)] F(A, p), F(B, A(p))), \text{if } \alpha(p) = \eta; \\ e, \text{if } \alpha(p) = \mu, \end{cases} \quad (3)$$

where $\alpha(p)$ is a condition; $A$, $B$ are operators; $e$ is an empty text.

According to the definition (3), the result of interpretation of the conditional operation will be the text of operator $A$, if the value of condition $\alpha(p)$ is true. The text of operator $B$ is outputted in the case if the condition $\alpha(p)$ is false. The text of conditional operation without changes is generated, if the value of $\alpha(p)$ was not computed, and the empty text will be the result, if an error occurred during the interpretation.

**Example 1.** Let us illustrate the application of AHS to transformation of a hybrid sorting algorithm. The algorithm reads a set of input numerical arrays and calls one of sorting sub-algorithms (*insertionSort, quickSort* or *mergeSort*) depending on the size of input array [8]. The arrays of size $n < MIN$ are sorted by *insertionSort*, the arrays of size more than *MAX* are processed by *quickSort* and arrays getting to an interval [*MIN*, *MAX*] are sorted by *mergeSort* algorithm. The regular scheme of the algorithm is

$$SORTING = INIT * \{[END\_OF\_SET]\ INPUT\_ARRAY(A) * \\ * ([n < MIN]\ insertionSort\ (A, n), \\ ([n > MAX]\ quickSort\ (A, n), mergeSort\ (A, n)))\}\ ,$$

where *INIT* is an initialization operator; *END_OF_SET* is a condition being true if all the arrays from the input set have been processed and false otherwise; *INPUT_ARRAY* is an operator that reads the array; $A$ is an input array and $n$ is its size.

Let it is in advance known, that the algorithm will be applied in conditions when the size of all input arrays is in a certain range, say, not less than *MIN*. Then the given RS becomes superfluous and for its transformation we will regard it as the hyperscheme with parameter $n$. At a stage of interpretation of hyperscheme *SORTING*, predicate $n < MIN$ takes value 0, whereas $n > MAX$ takes value $\eta$ ("not computed"). The condition $n > MAX$ cannot be computed because we do not have enough information about the value of parameter $n$ (we know only that $n \geq MIN$). Assuming that function $F$ is identical on a set of all basic operators and conditions of the hyperscheme, we will receive the reduced RS

$$F(SORTING, P_0) = INIT * \{[END\_OF\_SET]\ INPUT\_ARRAY(A) *$$

$$* ([n > MAX]\ quickSort\ (A, n), mergeSort\ (A, n)))\},$$

where $P_0$ is an initial state of information set.

Thus, by setting various values of parameter $n$ (in the considered case the value $n \geq MIN$ was set), it is possible to receive RS, optimum to usage conditions.

Paper [8] also describes more complicated hybrid sorting scheme, which selects an appropriate sorting algorithm depending on size and presortedness degree of input array. This algorithm also can be regarded as a hyperscheme.

## 3 Application of Hyperschemes to Representation of Derivation Algorithms in Generative Grammars

In this section algebra of hyperschemes is applied to representation of algorithms of inference control in structural design grammars (SDG) [1]. The approach is illustrated on an example of linear algebra algorithm.

SDG is a set of rules for generating the algorithms, specified in SAA (see Section 2). It is defined as a 5-tuple

$$G = (T, N, \alpha, P, D),$$

where $T = \Sigma \cup S$ is a set of terminal symbols, $\Sigma$ is a set of basic conditions, operators and data objects; $S$ is a set of separators, which are symbols of SAA operations, brackets etc.; $N$ is a set of non-terminal symbols (logical, operator and data object metavariables); $\alpha \in N$ is a start symbol; $P = \{u_i \to v_i \mid i = 1, 2, \ldots, k\}$ is a set of rules; $D$ is a derivation control algorithm. In this work derivation control algorithm is represented in algebra of hyperschemes.

**Example 2.** Let us illustrate the process of generating an algorithm scheme with usage of SDG and AHS on a parallel matrix multiplication task. The algorithm multiplies two rectangular matrices: $A = (a_{lj})_{M \times N}$ and $B = (b_{lj})_{N \times Q}$. The elements of a resultant matrix $C = (c_{lj})_{M \times Q} = A \times B$ are defined according to the formula

$$c_{lj} = \sum_{k=0}^{N-1} a_{lk} \cdot b_{kj}, l = 0,..., M-1, j = 0,..., Q-1,$$

where the elements of matrices are indexed beginning with zero.

In the parallel algorithm under consideration the elements of a resultant matrix are indexed according to the rule $n_{lj} = l \cdot N + j$. Computations are performed by $K$ processors in such a way that the first processor computes the first $\frac{M \cdot Q}{K}$ elements of

a resultant matrix, the second one processes the following $\frac{M \cdot Q}{K}$ elements, and so on. Thus, the initial matrix $A$ and final matrix $C$ are divided into horizontal blocks shown in Fig. 1. The processor with number $i$ multiplies block $A_i$ by matrix $B$ and receives the resultant block $C_i$. All blocks of matrices are also indexed beginning with zero.

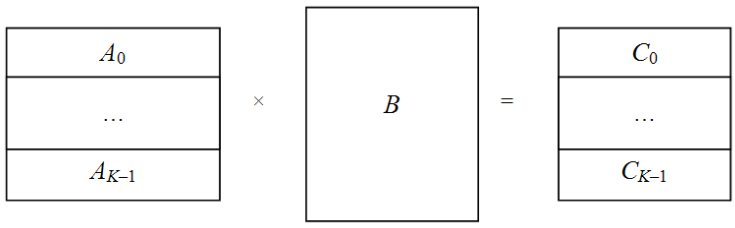


**Fig. 1.** Splitting of matrices into blocks in the parallel multiplication algorithm

The regular scheme of the parallel algorithm is

$$MatrixMultiplication(K) = START(K) *$$
$$* (Thread(A_0, B) // Thread(A_2, B) // \ldots // Thread(A_{K-1}, B)) *$$
$$* S(All\_Threads\_Completed) * FIN ,$$

where $START(K)$ is the operator of initialization of matrices and preparation for launching $K$ parallel threads; $Thread(A_i, B)$ is the operator, carrying out multiplication of $i$-th block of matrix $A$ by matrix $B$; $S(All\_Threads\_Completed)$ is the synchronizer operation (see Section 2), which delays the computation until all threads complete their work; $FIN$ is the final operator which outputs a resultant matrix $C$. The SAA scheme detailing the compound operator $Thread(A_i, B)$ is presented below.

```
Thread (Ai, B) =
  (start := M / K * i)*
* (end : = M / K * (i + 1)) *
* FOR (l) FROM (start) TO (end–1) DO
      FOR (j) FROM (0) TO (Q–1) DO
        (value : = A[l][0] * B[0][j]) *
        FOR (k) FROM (1) TO (N–1) DO
          (value : = value + A[l][k] * B[k][j])
        END OF LOOP *
        * (C[l][j] : = value)
      END OF LOOP
  END OF LOOP *
* CP(Thread_Completed(i)) ,
```

where $CP(All\_Threads\_Completed)$ is a control point operation (see Section 2), fixing the moment of completing the calculations in the thread with index $i$.

Let $G_1 = (T_1, N_1, \alpha_1, P_1, D_1)$ be a structural design grammar, intended for generating the class of schemes *MatrixMultiplication*(*K*). The SDG generates algorithms from the mentioned class with various number of parallel threades *K* depending on the resources available. The rules of grammar $G_1$ provide the generation of parallel threads $Thread(A_i, B)$ by changing the values of parameter *i* from 0 to *K*–1. The set of rules of SDG $G_1$ is the following:

$$m_0 : \alpha_1 \rightarrow START\ (0) * PRS1 * S\ (All\_Threads\_Completed) * FIN,$$

$$m_1 : \ PRS1 \rightarrow Thread(A_0, B)\ //\ PRS1,$$

$$m_2 : \left\| \begin{array}{l} Thread(A_i, B)\ //\ PRS1 \rightarrow Thread(A_i, B)\ //\ Thread(A_{i+1}, B)\ //\ PRS1 \\ START(i) \rightarrow START(i+1) \end{array} \right\|,$$
$$\text{at } i = 0, 1, \ldots, K-3,$$

$$m_3 : \left\| \begin{array}{l} Thread(A_i, B)\ //\ PRS1 \rightarrow Thread(A_i, B)\ //\ Thread(A_{i+1}, B) \\ START(i) \rightarrow START(i+2) \end{array} \right\|,$$
$$\text{at } i = K-2,$$

where $PRS1 \in N_1$ is an operator nonterminal, $A_i$ is a data nonterminal, and the rest are terminal symbols. The rules, which are applied concurrently, are grouped into rule sets $m_2$ and $m_3$.

The process of generation of the algorithm scheme according to given rules is the following. The rule $m_1$ forms a thread with number 0. Then the rule set $m_2$ recursively forms the next threades (at $i = 0, 1, \ldots, K-3$). The process completes with application of the rule set $m_3$ (at $i = K-2$).

The hyperscheme *MatrixMultHS*, which represents the derivation control algorithm for SDG $G_1$, is given below. It was built according to the method for constructing control algorithms for formal grammars, that was presented in [9].

*MatrixMultHS* = (*i*: = 0) * (*K*: = 4) *
* *START*(*K*) * *PRS*1 *
* *S*(*All_Threads_Completed*) * *FIN*,

*PRS*1 = *SELECT*
(
[(*i* >= 0) ∧ (*i* < *K*–1)] →
(*Thread*($A_i$, *B*) // *PRS*1),

[*i* = *K*–1] → *Thread*($A_i$, *B*)
) * *INC*(*i*) ,

where $K$ is a parameter of the hyperscheme (number of parallel threads); $PRS1$ is the compound operator that recursively generates a sequence of threads; $INC(i)$ is an increment operator that adds 1 to the value of index variable $i$. The execution of first three lines of the hyperscheme *MatrixMultHS* complies to application of rule $m_1$ of grammar $G_1$. The operators $START(K)$, $Thread(A_i, B)$ and $FIN$ are identical on information set $P$ and their performance consists in generation of the text of corresponding operator with the current value of variables $i$ and $K$. The execution compound operator $PRS1$ complies with application if rules $m_1$, $m_2$ and $m_3$. Setting specific values of parameter $K$ and subsequent interpretation of the hyperscheme allows to receive the schemes of matrix multiplication algorithms with corresponding number of threads.

For instance, as a result of execution of hyperscheme *MatrixMultHS* with value of parameter $K = 4$, we will receive the following parallel regular scheme:

$$MatrixMultiplication(4) = START\ (4) *$$
$$* (Thread(A_0, B)\ //\ Thread(A_1, B)\ //\ Thread(A_2, B)\ //\ Thread(A_3, B)) *$$
$$* S(All\_Threads\_Completed) * FIN\ .$$

For automating the construction of hyperschemes and generation of algorithms, the software toolkit was developed, which is considered in the following Section 4. The toolkit also supports generation of programs.

For verifying the efficiency of the developed parallel algorithm, corresponding multithreaded C++ program was generated. The program was executed on Intel Core 2 Quad processor (2.51 GHz). Fig. 2 shows its execution time in seconds. The speedup when executing it on 2, 3 and 4 processors was 2; 2.9 and 3.9 accordingly.

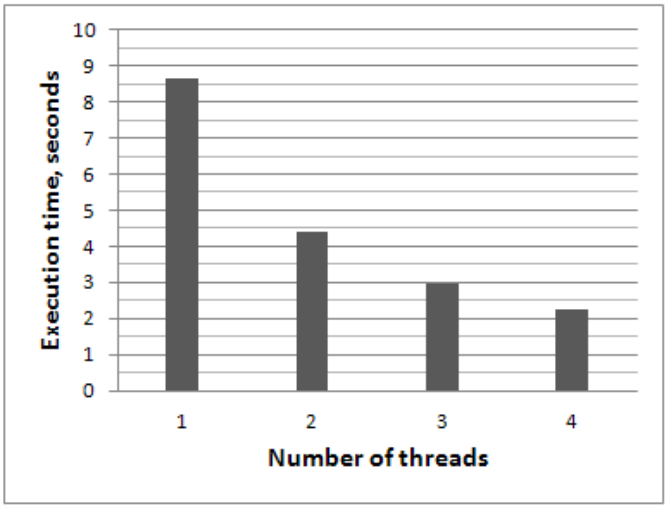


**Fig. 2.** Execution time of the parallel matrix multiplication program on 4-core processor; the size of input matrices is 1000 × 1000 elements

## 4 The Integrated Toolkit for Designing and Synthesis of Programs

The developed software tool (IDS) [1, 5] is based on usage of algebraic facilities, considered in Section 2, and it is intended for interactive constructing of algorithms and hyperschemes and generating of programs in target programming languages. Algorithms are constructed as syntactically correct programs ensuring the syntactical regularity of schemes. IDS integrates three forms of representation of algorithms at their designing: regular schemes, SAA schemes (textual representation of SAA formulae) and flow graphs [5]. The advantage of using textual representation of SAA schemes is the ability to describe algorithms in a form suitable for a human facilitating achievement of demanded quality of programs. At present time IDS supports the generation of programs in Java and C++ languages.

IDS toolkit consists of the following components (Fig. 3):

- the Constructor, intended for interactive designing of syntactically correct serial and concurrent algorithm schemes and generation of programs;
- flowgraph editor;
- interactive transformer of algorithm schemes based on application of algebraic equalities;
- generator of SAA schemes according to hyper-schemes;
- database, containing the description of SAA operations, basic operators and predicates in three forms mentioned above, and also their program implementations.

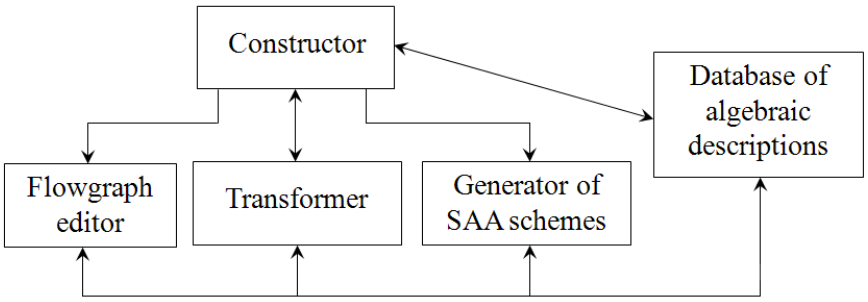


**Fig. 3.** The architecture of IDS toolkit

The Constructor of IDS toolkit is intended for top-down designing of algorithm schemes and hyperschemes by the superposition of SAA language constructs, which a user chooses from the list and which are considered as reusable components for construction of algorithms. The design process is represented by a tree of an algorithm. Fig. 4 shows a screenshot of the Constructor window with a matrix multiplication hyperscheme, that was considered in Section 3. The window contains three subwindows: the left upper subwindow includes a list of SAA operations, the subwindow on the right side contains a tree of an algorithm, and the third subwindow shows the text of SAA scheme.

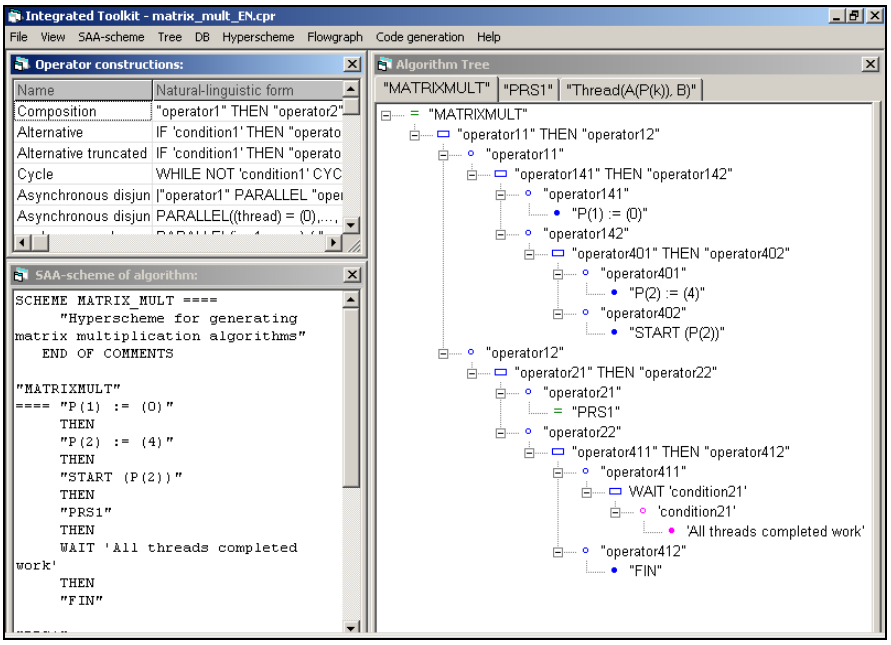


**Fig. 4.** The main window of the Constructor

On each step of the design process the Constructor allows a user to select only those operations, the insertion of which into the algorithm tree does not break the syntactical correctness of the scheme. The tree of an algorithm is then used for automatic generation of SAA scheme text, a flow graph and the program code in a target programming language. IDS was used for generation of multithreaded and message-passing parallel programs [1]; the application domain included sorting, search and linear algebra tasks. At present IDS does not have the tools for proving the correctness of program generation and a resultant code has to be verified by a user.

## 5 Conclusion

The approach to development of serial and parallel algorithms on the basis of usage of parameter-driven generation of algorithm schemes is proposed. The approach is based on algebras of algorithms and hyperschemes, and also structural design grammars. Hyperschemes are parameterized specifications that allow to receive the algorithms adapted to specific conditions of their use. The advantage of using algorithm schemes is that they are independent of a specific programming language and can be translated to an arbitrary language. The other advantage is that developed software tools are oriented on construction of algorithms and hyperschemes in the mode of interactive constructing, providing their syntactic regularity.

## References


1. Andon, F.I., Doroshenko, A.Y., Tseytlin, G.O., Yatsenko, O.A.: Algebra-Algorithmic Models and Methods of Parallel Programming. Akademperiodika, Kyiv (2007) (in Russian)
2. Bulyonkov, M.: Mixed Computation and Compilation: New Approaches to Old Problems. Theor. Comput. Sci., Vol. 71, pp. 209–226 (1990)
3. Doroshenko, A., Shevchenko, R.: A Rewriting Framework for Rule-Based Programming Dynamic Applications, Fundamenta Informaticae, Vol. 72, N1–3, pp. 95–108 (2006)
4. Doroshenko, A., Tseytlin, G., Yatsenko, O., Zachariya, L.: A Theory of Clones and Formalized Design of Programs. In Proceedings of the International Workshop on Concurrency, Specification and Programming (CS&P'2006), pp. 328–339 (2006)
5. Doroshenko, A., Tseytlin, G., Yatsenko, O., Zachariya, L.: Intensional Aspects of Algebra of Algorithmics. In Proceedings of the International Workshop on Concurrency, Specification and Programming (CS&P'2007)
6. Dybvig, R.K: From Macrogeneration to Syntactic Abstraction. Higher-Order and Symbolic Computation, Vol. 13, pp. 57–63 (2000)
7. Mayr, R., Nipkow, T: High Order Rewrite Systems and Their Confluence. Theoretical Computer Science, Vol. 192, pp. 3–29 (1998)
8. Yatsenko, O: On Application of Machine Learning for Development of Adaptive Sorting Programs in Algebra of Algorithms. In Proceedings of the International Workshop on Concurrency, Specification and Programming (CS&P'2011), pp. 577–588 (2011)
9. Yushchenko, E.L., Tseitlin, G.E., Galushka A.V.: Algebraic-Grammatical Specifications and Synthesis of Structured Program Schemas. Cybernetics and Systems Analysis, Vol. 25, N 6, pp. 713–727 (1989)