Bull. Nov. Comp. Center, Comp.Science, 14 (2001), 43–58
© 2001 NCC Publisher

Basic constructions of models in WinALT[*]

M.B. Ostapkevich, S.V. Piskunov

The methods and tools of model construction with fine-grain algorithms and structures are considered and demonstrated by typical examples. The usage of the system for models with complicated structure and large amount of data is presented.

Introduction

The WinALT is a simulating system of fine-grain algorithms and structures. Its overall description is presented in [1–3]. From a user's point of view the main WinALT window has accustomed appearance of a Windows application. Its interface is recognizable and intuitively clear, which undoubtedly ease the learning of the system. A model created by the means of the system can be distinctively divided into two tightly interacting parts: graphic and textual. Each of them is supported by relevant subsystem of WinALT.

The graphic subsystem [4] along with its usual functions such as visual representation of source, intermediate and resulting data in a model, iconic view of tools and services, multiwindow document interface implements a number of specific functions. These are determined by the main destination of the system: being a tool for fine-grain algorithms and structures research. Among such functions the construction of graphic images for commands and the visualization of their application can be listed.

The language subsystem [5] supplies a user with a set of tools for construction of textual part of model: the statements of a structured programming language similar to Pascal, the means for creation of libraries and finally the most important ones, the tools for concise representation of distributed in space parallel computations. We shall refer to the textual part of the model as a simulating program in the further text.

The papers mentioned above are targeted on the description of the system architecture, the graphic interface and the language rather than the description of model construction within the system. Only [6] could be mentioned as an exception. This article considers the two-level procedure of aggregative model construction, but it is assumed in it that a user is already familiar with basic WinALT functions and is capable of creating simple models (those of the first level).

[*]Supported by the Russian Foundation for Basic Research under Grant 99-07-90422.

The *goal* of the article is to aid a user to learn the basic functions of WinALT in the process of model construction up to the level sufficient for capability of building his own model with the help of standard samples and user's documentation.

The main attention is drawn in the article to simultaneous usage of all functions and tools provided by a system when creating a model. The same service can be provided in several ways, only the most evident of which shall be mentioned.

1. The structure of a model

1.1. The general structure of a model. It is rational to represent a separate model by a separate project. In general, the mechanisms of project construction are similar to those in such Windows application as Microsoft Visual Studio [7]. A project is created by the **Project** tab in a dialogue from **File/New** menu of the WinALT frame window. The construction of a project results in creation of a directory that has the same name as the project in a certain part of the directory tree. Inside the directory a file with `.wap` extension appears. This file is intended to store the list of files that belong to the project. After a project has been opened, a user gains access to all the tools and can create and modify a model. Visually a model lies in windows of two types.

The window of the first type, which has an associated file with `.3do` extension, is created by selecting the string in **Cellular Objects** tab of the dialogue activated by **File/New** menu item. A lot of such windows could be created. These windows contain objects of a model. The extension of an object file depends on the format of stored data. For example, a file that contains a pixel image has `.bmp` extension. The extension of default type object file is `.axt`. The Boolean and integer types both have `.ast`, while floating point object type has `.aft` extension and so on. It is up to a user to decide which object located in which window. Objects can be dragged from one window to another. The same object can be located in more than one such window.

The second type of window is represented by files with `.src` extension. Such window can be created by selecting the string in **Simulated Program** tab of the **File/New** menu item dialogue. This window should contain a syntactically correct version of a simulating program after a model has been created. The windows title bears the name of the program the window contains. If a user wants to create more than one simulating program in single model, he has to create the same number of windows of this type. This might be useful for example when more than one program process the same set of objects or their sets of objects have a certain intersection.

1.2. The structure of the model graphic part. Graphic objects can be created and edited with the help of **Tools** toolbar (Figure 1). The tools presented in this toolbar are called **Arrow**, **Creator**, **Magic Wand** (from left to right). **Creator** instrument is designed for cellular array and template creation. A user has to press the left mouse button and expand the rectangular area so as to obtain the desired location and visual size of the creating object. After the mouse button released, **New Object** dialogue box appears. This dialogue allows the setting of object size for three dimensions, object name including the type of object. A user may select one of the following standard predefined object types: `default`, `boolean`, `integer`, `float`, `string`, `character`. `Default` type allows to set values of all the WinALT types to any cell. And besides, each cell may be declared empty by setting its type to `void`. Also, a cell may be named by an identifier. Such assignments can be accomplished by **Magic Wand** tool. When it is selected, a user may see an object with input focus that points to the current cell in an object. By moving the focus, different cells may be initialized with a value and a type. Cell properties could be seen at the status line at the bottom of WinALT main window. **Arrow** tool let select an object or a group of objects, move the selected objects in a sheet, change their visual sizes. The principal difference between an object and a template is semantic rather than formal. Arrays are intended for data storage while templates set right and left parts of substitution commands [8] in the graphic form. As usual, arrays have `boolean`, `integer`, `float`, `string`, `character` types. `Void` type appears in array only in exceptional cases. Cells are distinguished by colors. Everything is different with templates, the main object type for them is `default` and `void` type is widely used for its cells. Empty cells are denoted by white crossed squares. But it is worthy to mention that it is allowed to utilize the same cellular object both as an array and as a template within the same model. For further explanation of model construction methods let us divide templates into three types. The templates of the first type do not contain named cells, there are only colored or `void` cells in such objects. The templates of the second type contain only `void` cells, some of which are named. Finally, the templates of the third type contain both colored and empty cells some of which are named.

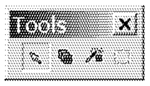


Figure 1

1.3. The structure of a simulating program is traditional. It contains the following types of parts: a library call, a template description, a variable declaration, a constant definition, a procedure, a function and the main program body. Any type of fragment except for the main body could be omitted. Or vice versa it can be repeated more than once. The blocks could be located in a random order, but of course, the definition of a term must precede its usage. For example, the function declaration must lie before its call. The fragments and their items are denoted by keywords with evident meaning. For example, that could be demonstrated by template declaration, which is a statement specific for WinALT. This fragment lies within `object-end` block. The element of such block is a command template description, which contains variable names. It contains a `pattern` keyword and an identifier that specifies template. Due to the fact that WinALT language possesses quite a poor set of its own operations, a user is provided with a means for data transformations with the help of libraries, which can be imported into a WinALT program by `import` or `use` statements. These ones differ in the way they load functions from libraries. `Use` adds names indirectly that means that function calls must include a library name followed by two colons. `Import` allows using function names without explicit specification of library where they are located. Thus, if two libraries contain functions with the same name, at least one of them has to be imported indirectly. The inclusion of external modules written in WinALT language is done by a preprocessor command `include`. The syntax of all the three operators is similar: a keyword is followed by a name of library. Nearly any more or less complicated model contains the import of `dcms_gio` and `standard` libraries. `Dcms_gio` contains functions for value input/output for WinALT standard types: `boolean`, `integer`, `float`, `string`, `character`. `Standard` implements functions for type cast, object management, benchmarking and so on. Two more libraries ought to be mentioned, as they will be useful for a user from the very first step of system learning. These are `Altio` and `Vmisc`. The former has operations for console input/output. For example, there are string and integer value input/output. The console input/output is done in a dedicated for this purpose dockable window inside the main WinALT window. `Vmisc` is composed by functions for the 2D vector graphics (drawing of primitives, color settings, line and fill style, text output).

The libraries are located in `acllib` and `bin` directories inside WinALT root directory. The set of functions inside a library can be listed by issuing a command `dumpacl.exe <`*library name*`> >> <`*output file*`>`.

The declarations of variables and constants are similar to those in high level languages, thus it will not be mentioned in the further text. The same remark could be done for function and procedure usage.

2. The basic constructions of typical fine-grain models

The basic constructions will be introduced with gradually increasing complexity of simulating object. We shall consider mainly the construction of models with the 2D cellular arrays and templates, though of course the system has the capability of processing objects with the maximal number of dimension equal to three. We select the 2D because of its simplicity and improved visual perception in comparison with the 3D. But if a user learns the construction of the 2D models, it will not be a great problem to commence the construction of the 3D models.

2.1. Models with one cellular array and templates of the first type. Let us commence with the simplest case when a simulating object is a fine-grain structure represented by an array (for example, the 2D). It consists of cells with the same type. A cell contains a memory element. The time is discrete. A cell functioning is described by a finite set of rules. A cell determines its new state and the new state for some of its neighbours (`O`) applying these rules to the current state of neighbour cells and one of its own (`I`).

Let us create cellular objects which are required for a model of such a structure.

First of all, we create a cellular array that contains the cell states of the structure. A user may construct an array and set the initial values of its cells as it was described in 1.2. Depending on which alphabet of cell states a structure cell has, a user chooses between the existing types of cellular arrays and WinALT value types.

The creation of templates is commenced considering the case when a structure cell is a finite automaton with inputs and outputs and the rules are state transitions. In this case, the templates of the first type could be used. A template of the first type is defined by the states of a cell and its neighbours. The default visualization mode in WinALT depicts cell states by colors. The size of template is selected so that all the relevant cells reside inside the rectangular area of the minimal dimensions. All the cells that are ignored but included in the rectangle of the template have to be initialized with `void` type. Each rule has two associated templates: one for I set and another for O set. The former is called input template, while the latter is output template of the rule. If the two sets coincide, the sizes for both templates are the same. Otherwise the sizes are different. The samples of templates are depicted in Figure 2.

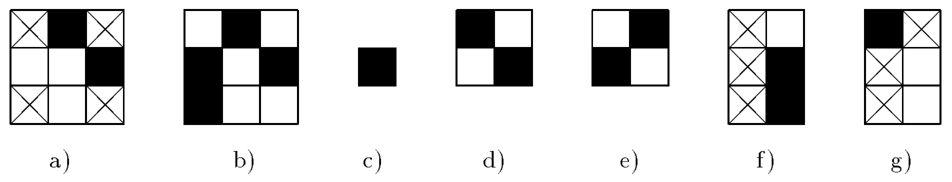


Figure 2. Samples of typical templates: a), b) left-hand part, c) right-hand parts for rules with the von Neumann and the Moore automata, d), e) both parts of rules for the Margolus automata, f), g) both parts of symbolic substitutions

It is assumed in WinALT that the top left cell of the template is the anchor that is applied to any cell of the cellular arrays. For pairs of objects it might be required to adjust their reciprocal positions. This could be accomplished by insertion of void cells in one of the templates. But a better way is to issue `shift` operator that will be discussed later. A sample of such an adjustment is depicted in Figure 3.

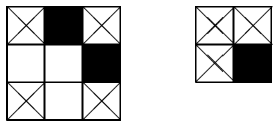


Figure 3. The adjustment of classic cellular automata templates

The samples of structures which could be simulated by the templates described above are the classical cellular automata [9], cellular automata with the Margolus neighbourhood [10], symbolic substitution algorithms [11], or parallel substitution algorithms [8], if there are no functional symbols in its record.

After all the required graphic object have been constructed, a simulating program can be written.

To simulate one step of a fine-grain structure functioning, a simulating program has to include operators that preform tiling for each cell in a cellular array by the left-hand templates. Whenever all the non void cells of the cellular array coincide with cells of applied template, they are substituted by ones from the right-hand template. In the language, this tiling for one rule is represented by the following bunch of the following statements:

```
in <cellular array name>
   at <left hand template>
      do <right hand template>
```

The pair of `at-do` operators is a command of substitution [8].

Note 1. If the adjustment of templates is required, `shift` operator is placed after `at`. `Shift` contains the name of template and three coordinates in brackets as parameters. The three coordinates denote three shifts for each axis. The `shift()<`*right hand template name*`>(1,1,0)` operator performs the same adjustment as depicted in Figure 3.

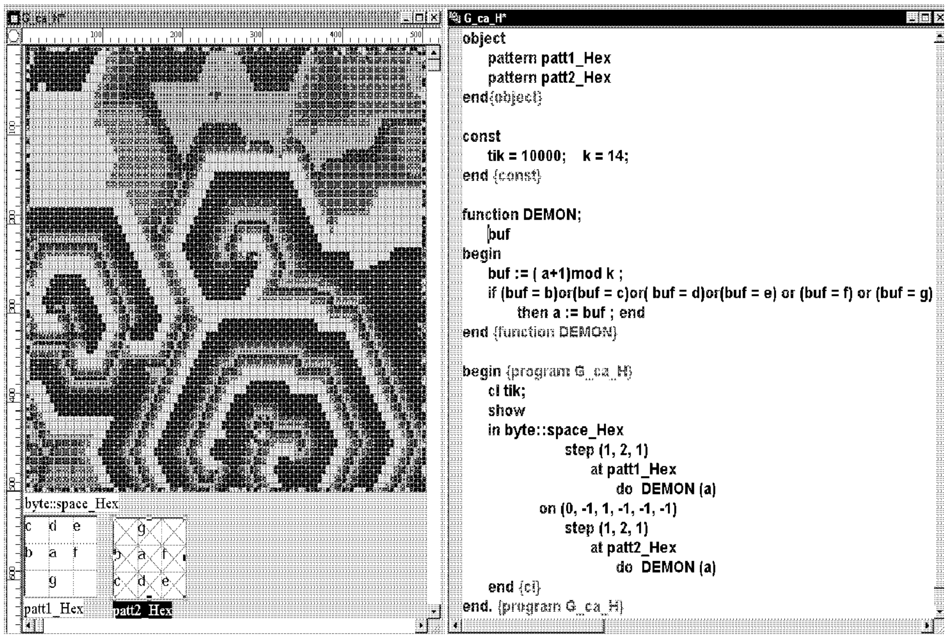
The number of such constructions coincides with the number of transition rules in the table of transitions. Their order is random. A record can be more concise by excluding all `in` statements except the first one. Let us denote such a record as R. To specify that all the changes of cell values are synchronous, the record R is placed within so called synchroblock. There are three types of such blocks in the language. A user chooses one depending on the number of iterations and the condition of termination. If only one step is to be executed, `ch <`R`> end` synchroblock is used. If k ($k \geq 1$) iterative synchroblock `cl` k`; <`R`> end` is the most appropriate one. Finally, the third type of synchroblock is implemented for the case when a model has to repeat steps until no more changes of cell values occur (`ex <R> end`). It can iterate eternally. To create a valid program, the synchoblock must be placed within `begin-end.` block. The resulting program consists only of the main body. Though being rather simple it is sufficient for many fine-grain models simulation (such as classical cellular automata, cellular structures with some non-classical neighbourhood [8, 11]). But a program with such a structure cannot simulate many other models, such as an automaton with the Margolus neighbourhood. Here a more complicated tiling is required, as odd and even cells are handled differently. This sort of tiling could be accomplished by `on` and `step` operators. `On` is placed after `in` and is used for selection of a rectangular area inside the cellular array selected in `in`. `On` takes six parameters in brackets. Parameters are separated by comma. The first pair sets the range for x axis, the rest are for y and z respectively. The value -1 denotes that the border coincides with the border of an array. If `on` is included into the bunch, the tiling is made only for the selected rectangle. Operator `step` sets the step of tiling. It takes three parameters in brackets with comma as a separator. The first one sets step for x axis, the second and the third are for y and z respectively. The main body for the Margolus neighbourhood might look like this:

```
begin
    ex
        ch
            in <cellular_array_name>
                step (2, 2, 1)
                    <R>
        end
        ch
            in <cellular_array_name>
                on <1,-1,1,-1,-1,-1>
                    step (2, 2, 1)
                        <R>
        end
    end
end.
```

The simulation of the Margolus automaton is based on the alternation of two grids, which is accomplished by placing two `ch` synchroblocks sequentially into one `ex` synchroblock. The odd grid is imitated done via shift of the region for which the tiling is constructed by one cell to the bottom and to the left with the help of `on` operator.

2.2. Models with one cellular array and templates of the second type. The templates of the first type have rather wide application both in digital device [12] and physical [9, 13] models, as many types of their data transformations can be performed by finite automata. But in the most of practically useful models the state transition tables have a huge number of entries thus leading to the big size of a program's main body. As an example, the Griffite (its description is given in the sample below) automaton transition table could be considered. In the case of cellular neural network with continuous sigmoide function such a table cannot be constructed. The solution is to utilize templates of the second type proposed by the system. These templates being a part of the graphic subsystem allow using functions in the WinALT language. The interconnection of templates and functions is established by a couple of `at` and `do` operators where the former contains the template name as a parameter while the latter specifies the associated function. All or some of template cell names are treated as function variables. The ones that are initialized within a function must be placed in the list after the function name in the call statement. This list should be within brackets. Its items are separated by semicolons or commas. The template variable names are local in the sense that the same names can be used in other templates. A function may have as well its own local variables for the sake of particular implementation convenience. These variables are defined immediately after the name of the function.

Example 1. The Griffite automaton model on hexagon grid is depicted in Figure 4. Each cell of `byte::space_Hex array` can have one of k states, numbered from 0 to $k-1$. Each cell alters its state according the following rule (`Demon` function): if the central cell named `a` in `patt1_Hex or patt2_Hex` templates has s state at the current step, and the state of either `b`, `c`, `d`, or `e` cell is greater by one, the value of the central cell becomes



```
object
    pattern patt1_Hex
    pattern patt2_Hex
end{object}

const
    tik = 10000;    k = 14;
end {const}

function DEMON;
    buf
begin
    buf := ( a+1)mod k ;
    if (buf = b)or(buf = c)or( buf = d)or(buf = e) or (buf = f) or (buf = g)
        then a := buf ; end
end {function DEMON}

begin {program G_ca_H}
    cl tik;
    show
    in byte::space_Hex
                step (1, 2, 1)
                    at patt1_Hex
                        do  DEMON (a)
            on (0, -1, 1, -1, -1, -1)
                step (1, 2, 1)
                    at patt2_Hex
                        do  DEMON (a)
    end {cl}
end. {program G_ca_H}
```

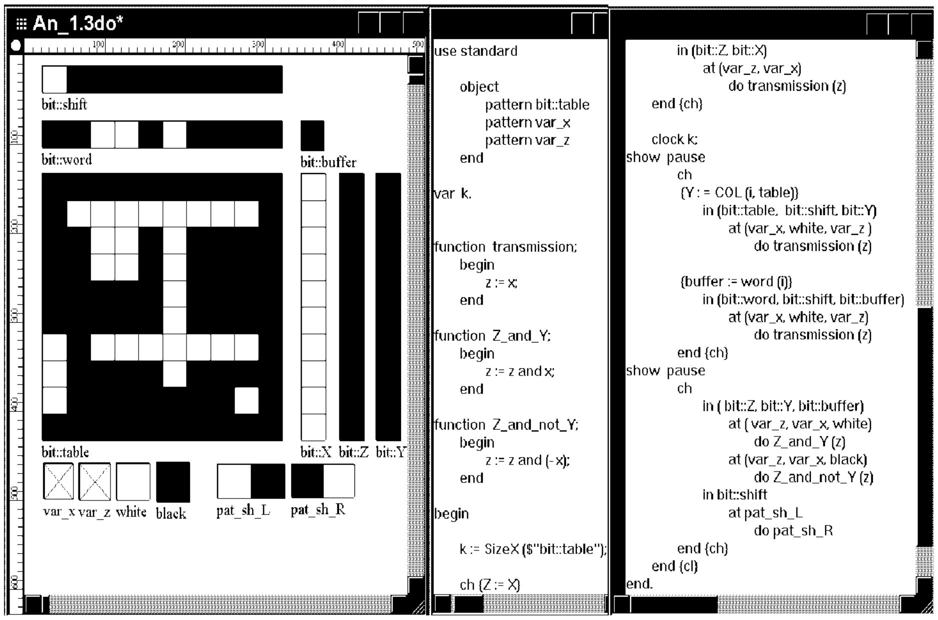
Figure 4

$(s+1)$ mod k at the next step. The implementation of hexagon grid above the rectangular is based upon the usage of pair of templates and `on` and `step` operators.

Note 2. An auxiliary operator `show` is used in the simulating program in Figure 4. It redraws all the objects with their current cell states. Of course, more than one such operator can be placed in a program. There is one more auxiliary operator `stop` intended to suspend the program. In the graphic edition, a program execution can be resumed by pressing resume button at execution toolbar (F1).

2.3. Models with one cellular array and templates of the third type. These templates unlike ones of the second type are more demonstrative. The text of the function linked with such a template is simpler as the values of coloured cells need not to be verified within its body. There are no essential differences between these templates and those of the second type.

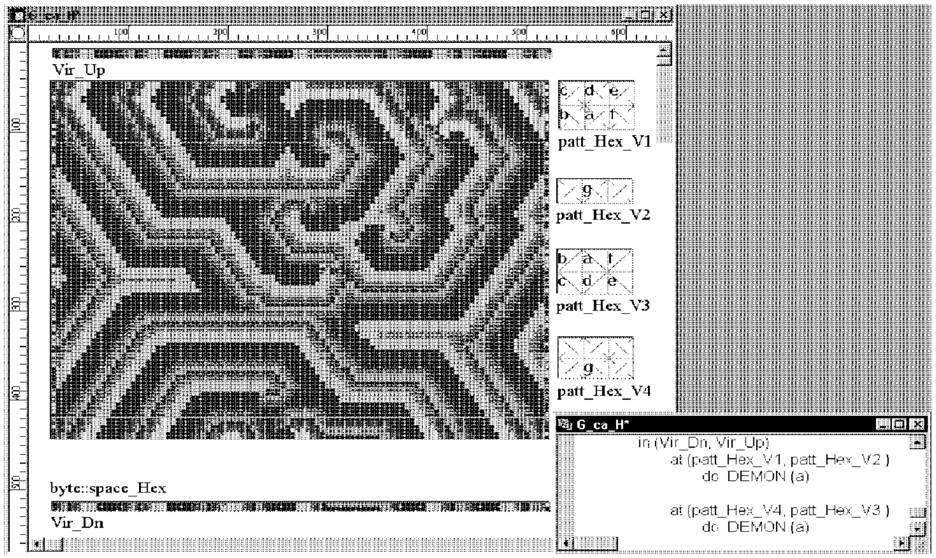
2.4. Models with several cellular arrays and templates of all types. The models with one big cellular array are typical for fine-grain structures that simulate physical phenomena or cellular automata behaviour in the finite part of infinite discrete space. But a structure that has more than one such array can be easily imagined. For example, an associative processor may consist of one 2D bit array and several 1D slices. The 2D matrix is an associative memory divided into strings that contain numbers. The 1D slices are lines of simple processor nodes, which process simultaneously the columns of the matrix. For the construction of such sort of structures the WinALT gives the means for vector processing. The tuple `in`, `at`, `do` is a part of the means. When in vector mode these operators contain a list of objects names in brackets, the items as usual are separated by comma or semicolon. If `at` list contains templates of the second or third type, `do` must contain a single name of a function. Such function can use named cells from any object in `at` vector as its variables. If the list contains only templates of the first type, `do` may be followed by similar template list. The syntax requires that the number of list items in both lists should be equal, otherwise the longer list will be truncated to length of the shorter one. However, a user may use an artificial trick. A name of once void cell template could be placed to the position, which is to be ignored. To enable the usage of templates with different sizes it is assumed that the first array in `in` list is the master and other are slaves. If the slave array is greater or equal to the master, the coordinates within the vector substitution are the same. If it is smaller by a certain axis, then its coordinate is calculating by taking the module of master coordinate division by the slave array size for that axis. This way of coordinate transformation allows to list arrays with different



```
use standard

    object
        pattern bit::table
        pattern var_x
        pattern var_z
    end

var k.


function transmission;
    begin
        z := x;
    end

function Z_and_Y;
    begin
        z := z and x;
    end

function Z_and_not_Y;
    begin
        z := z and (-x);
    end

begin

    k := SizeX ($"bit::table");

    ch (Z := X)
```

```
            in (bit::Z, bit::X)
                at (var_z, var_x)
                    do transmission (z)
        end (ch)

        clock k;
show pause
            ch
             {Y : = COL (i, table)}
                in (bit::table,  bit::shift, bit::Y)
                    at (var_x, white, var_z )
                        do transmission (z)

             {buffer := word (i)}
                in (bit::word, bit::shift, bit::buffer)
                    at (var_x, white, var_z)
                        do transmission (z)
            end (ch)
show pause
            ch
                in ( bit::Z, bit::Y, bit::buffer)
                    at ( var_z, var_x, white)
                        do Z_and_Y (z)
                    at (var_z, var_x, black)
                        do Z_and_not_Y (z)
                in bit::shift
                    at pat_sh_L
                        do pat_sh_R
            end (ch)
        end (cl)
end.
```

Figure 5



```
            in (Vir_Dn, Vir_Up)
                at (patt_Hex_V1, patt_Hex_V2 )
                    do  DEMON (a)

                at (patt_Hex_V4, patt_Hex_V3 )
                    do  DEMON (a)
```

Figure 6

dimensions in the same list. If on follows in the selection is done in the master array. The selection in slaves is done in accordance with the rule described above. Step sets the same steps for all items in vector.

Example 2. An imitational model of fine-grain structure that implements a search algorithm by a key from [14] is depicted in Figure 5. A key is a bit word kept in bit::word register. The search of words that coincide with the key is performed in the strings of bit::table array. The words that have coincidence with the key are marked in the respective bits of bit::Z register unless they are masked by zero in bit::X register. The algorithm of search is set by Z_and_Y and Z_and_not_Y functions, which use x and y cell names from templates var_x and var_y as variables defined for Z and Y registers. The selection of a function at *i*-th step depends on the state of buffer cell, that keeps *i*-th digit of bit::word register. The *i*-th digit is selected with the help of bit::shift register.

The tool for virtual cellular array creation can also be assumed as a means for parallel computations. A virtual array is a subarray of another array. A virtual array is created via Create Virtual Object window activated from Virtual menu of the main window if a parent array was selected by Arrow tool before. The values of cells in both arrays are changed synchronously. A virtual array plays the same role as on operator for its parent array. But in the case of vector operators its role is somewhat different. Unless on, which can only select parts of arrays, a virtual array can itself be placed in a vector list. More than one virtual arrays could be defined for the same parent array.

Example 3. Virtual cellular arrays could be used to emulate infinite cellular space for a certain dimension. Let us close together the ends of byte::space_Hex array from Example 1 by x axis so as to form a cylinder. The required changes in the model are shown in Figure 6. Patt_Hex_V1, patt_Hex_V2, patt_Hex_V3, patt_Hex_V4 templates and Vir_Up, Vir_Dn virtual arrays are added to the graphic part. Two virtual arrays are located in the two top and two bottom rows of the parent array respectively. For the sake of simplicity, it is assumed that the number of cells along y axis in byte::space_Hex array is even. The operator tuples shown in the picture on the right are inserted in cl-end synchroblock.

3. Auxiliary methods and tools for fine-grain model construction

Previously, the basic types of program structures were considered. They are sufficient for many interesting but rather simple models. Nevertheless, it is evident that the set of models is virtually unexhaustible and the models can be quite complicated. So now we would like to draw user's attention to such methods and system tools of model construction, that are aimed to overcome the problems of a model complexity. We have to take into consideration that there are several kinds of model complexity. Namely, these are:

1) the amount of processed data;
2) the complexity of simulating program structure;
3) the difficulty of cell neighbourhood definition;
4) non-trivial search of the area where a substitution command may be applied; and
5) the complexity of functional transformation of data within a cell of cellular array.

3.1. The support for construction of models with huge amount of data. As usual the real-life problems contain a huge amount of transformed data. Their simulation is time consuming. This is the case when the graphic user's interface, which is created for the interactive mode, gives no advantages in comparison with the traditional text console. *Au contraire*, the usage of the GUI version lacks the efficiency of console and slows down the speed of simulation. Thus, a user is provided with a console version of WinALT. A user can lunch it from any console mode file shell as Far Commander by issuing the following commands:

1) adding the WinALT binary directory to the PATH variable so as to avoid printing the full path to the console version executable file;
2) changing the current disk and the current directory to the disk and the directory of project to be simulated; and
3) printing the command line, which contains the name of console executable "xaltcon.exe" and the name of source program file to run, for example, it would be written in the DOS shell prompt:

```
c:
cd \winalt\projects\MyProject
xaltcon.exe MyProgram.src
```

A user is recommended to divide the construction of model into two stages. First, a model is designed and debugged with the small objects interactively in the GUI version. When debugged a program could be executed in the console mode with data of real size. To ease the visualization in the text console, a Console library is implemented. In contains the procedures for colored text output and cursor positioning. A sample object viewer is implemented in WinALT. Its interface is similar to that of Far Commander. The sample is located in *ncviewer* directory within samples folder.

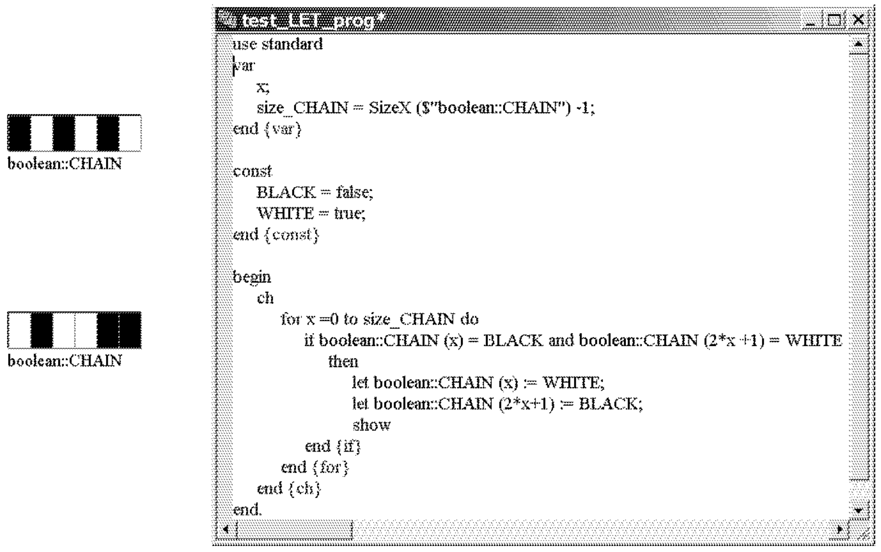
3.2. The structurization of simulating programs. A program that resides within a single `.src` file could be structurized by splitting it into several functions and procedures. But even in this case the source text might become quite vast and thus hardly readable. In this situation, a user can divide a monolith program into modules kept in separate files. The files could be inserted into the file that contain the main program body by the preprocessor `include` statement mentioned in Subsection 1.3. The division into modules is demonstrated by the following sample. Let a certain file `main.src` contain the source with `a` and `b` procedures. The former prints "WinALT!" text, while the latter call `a`:

```
1    use altio
2    procedure a()
3    begin
4       WriteStringLn($"WinALT!")
5    end {a}
6    procedure b()
7    begin
8       a()
9    end {b}
10   begin
11      b()
12   end.
```

This file could be broken into two. The bf a procedure is placed in an auxiliary file, e.g. `auxiliary.inc`. It contains strings 1–5. The text of main file contains `include auxiliary` as the first string and strings 6–12 as the tail. Only the main source file has to be added to project. An auxiliary file could be placed anywhere in the file system, though the most preferable place is the directory of the project.

Also a user should take into consideration that there are virtually no limits in the system for the nested synchroblocks and joint usage of the operators of the first and second tiers. This mixture allows constructing parallel-sequential composition of synchronous cellular array transformations of different kinds. Let us mark as well that a parameter of `cl` can be an expression that controls the number of iterations dynamically when executing thus simplifying the structure of a program.

3.3. The means of non-local neighbourhood template construction. The templates considered in Section 2 had only local neighbours. To avoid this constraint, an operator for the synchronous assignment should be used. It has similar syntax as the traditional assignment in WinALT with the only exception. A keyword `let` must precede the assignment itself. Unlike the ordinary assignment which alters the value immediately, the synchronous version changes is when leaving the synchroblock it resides.



```
use standard
var
    x;
    size_CHAIN = SizeX ($"boolean::CHAIN") -1;
end {var}

const
    BLACK = false;
    WHITE = true;
end {const}

begin
    ch
        for x =0 to size_CHAIN do
            if boolean::CHAIN (x) = BLACK and boolean::CHAIN (2*x +1) = WHITE
                then
                    let boolean::CHAIN (x) := WHITE;
                    let boolean::CHAIN (2*x+1) := BLACK;
                    show
            end {if}
        end {for}
    end {ch}
end.
```

Figure 7

A global or local variable, cell name in a template or a cell can be the destination of this sort of assignment. A cell is specified by an array name and three coordinates separated by comma inside round brackets. A separate coordinate is an expression. Particularly it could be an integer constant or a variable. A source of an assignment is an expression. Its syntax is much similar to that of Pascal. It is the usage of expressions and operators of the first tier, especially conditional operator and loops that lets select a subset of cells within an array, which reciprocal location depends, for example, on a coordinate value in an array of index variable, a counter of iterations, an index variable and so on. Let us demonstrate that by a sample (Figure 7).

3.4. Dynamic selection of template anchor cells and regions of substitutional command applicability. The usage of expressions as parameters in `step`, `on`, and `shift` operators helps to overcome the complexity of neighbourhood cell definition in a substitution. For example, let a 2D array `A` with size 2×3 be defined. At the same time, any bundle of `in-at-do` can include `on(A(0,0), A(0,1), A(1,0), A(1,1), A(2,1), A(2,1))` operators. In this case, along with changing of `A` array cell values, the region of `at-do` command applicability is altered.

3.5. The construction of hierarchical models. A user may separate a single and complex transformation performed within a cell into a series of simpler ones, which have a hierarchical dependence. This could be accomplished by using synchroblocks and calls of functions within `do` operators inside a body of a substitutional command. Such an action may be interpreted as a construction of a model that consists of several nested cellular structures.

4. Conclusion

The further acquirement of methods and tools of model construction can be continued by using the sample models that are installed along with WinALT. These examples reside in the samples folder within home WinALT directory.

References

[1] Piskunov S.V. WinALT – a simulation system for computations with spatial parallelism // NCC Bulletin, Series Comp. Science. – Novosibirsk: NCC Publisher, 1997. – Issue 6. – P. 71–85.

[2] Beletkov D.T., Ostapkevich M.B., Piskunov S.V., Zhileev I.V. The tools of language and graphic interface of a simulating system for computations with spatial parallelism // Proc. of the VIth Intern. Workshop "Distributed Data Processing". – Novosibirsk, 1998. – P. 228–232 (in Russian).

[3] Beletkov D.T., Ostapkevich M.B., Piskunov S.V., Zhileev I.V. WinALT, a software tool for fine-grain algorithms and structures synthesis and simulation // Lect. Notes in Comput. Sci. – 1999. – Vol. 1662. – P. 491–496.

[4] Beletkov D.T. The graphic construction of computer 3D models of cellular algorithms and structures // Proc. Conf. of Young Scientists / Institute of Computational Mathematics and Mathematical Geophysics. – Novosibirsk, 1998. – P. 3–13 (in Russian).

[5] Ostapkevich M.B. The WinALT system language tools // Proc. Conf. of Young Scientists / Institute of Computational Mathematics and Mathematical Geophysics. – Novosibirsk, 1998. – P. 182–194 (in Russian).

[6] Beletkov D.T., Zhileev I.V. Construction of models of computing structures with fine-grain parallelism in system WinALT // This issue. – P. 1–6.

[7] Kruglinski D.J. Inside Visual C++. – Microsoft Press, 1996.

[8] Achasova S.M., Bandman O.L., Markova V.P., Piskunov S.V. Parallel Substitution Algorithm. Theory and Application. – Singapore: World Scientific, 1994.

[9] Codd E.F. Cellular Automata. – New York; London: Acad. Press, 1968.

[10] Toffoli T., Margolus N. Cellular Automata Mechanics. – Massachusetts Institute of Technology, 1987; Russian Translation. – Moscow: Mir, 1991.

[11] Brenner K.H. Programmable optical processor based on symbolic substitution // Applied Optics. – 1988. – Vol. 27, № 9. – P. 1687–1691.

[12] Fet Ya.I. Parallel Processing in Cellular Arrays. – Taunton, UK: Research Studies Press, Ltd., 1995.

[13] Wolfram Stephen Theory and Applications of Cellular Automata. – Singapore: World Scientific, 1986.

[14] Nepomniaschaya A.S. Language STAR for associative and parallel computation with vertical data processing // Proc. Intern. Conf. "Parallel Computing Technologies". – Singapore: World Scientific, 1991. – P. 258–265.