Bull. Nov. Comp. Center, Comp. Science, 9 (1998), 93–106

The open architecture of WinALT

M. Ostapkevich

The necessity of the open architecture for fine-grained parallel model simulating system (WinALT) is discussed. The description of WinALT open architecture and external module interfaces is given. WinALT consists of the language and graphical user's interface subsystems and the kernel. The extensibility of WinALT is implemented in the kernel by a number of interfaces. The principal ones are object file format support interface and those of language and graphical subsystem extensibility. A number of samples is given to clarify the usage of all these interfaces.

1. Introduction

1.1. Fine grain simulating system WinALT

WinALT is a fine grain simulating system. It is intended to simulate complex dynamic systems, such as digital electronic devices (associative or systolic structures and 3D pipeline units), physical and biological systems, which are represented by cellular automata, neural and cellular-neural networks. It is based on the concept named the Parallel Substitution Algorithm (PSA) [1]. The general description of WinALT and the justification of its main features was presented in [2]. In this article the detailed description of the WinALT open architecture is given. The system has language and graphical means to execute, debug and examine cellular algorithms. Unlike its ancestors the system has the ability to be changed by a user, as it has the open architecture. In the article the brief description of open systems is given. The advantages of the open system concept for a Parallel Substitution Algorithm based simulating system are listed. The means of WinALT extensibility, scalability and interoperability are discussed in details. The description of WinALT interfaces and program samples, which use these interfaces, is given.

1.2. The concept of the open architecture

N. Wirth in [3] mentioned *monolithic design* of applications as one of the main causes of code size growth tendency. The term *monolithic* implies that an indivisible and inextensible set of binary modules exists and includes all the functions of a system, either important or useless for a certain user. A user cannot include new modules or exclude old ones out from such a system.

An open system is the exact antithesis of a monolithic one and it overcomes the faults mentioned above. The basic minimal set of functions is represented by a reduced set of modules and does not occupy huge space in a retrieval system. Such a system with the ability to include and exclude external modules may be tailored by a user.

In [4] the following distinguishing features of an open system were listed:

1. *Extensibility and scalability* is the possibility of a new function addition or an existing function modification while the rest functions remain unchanged.
2. *Portability* is the ability to be reimplemented on another platform without noticeable differences for a user.
3. *Interoperability* is the ability to communicate with other software systems.
4. User friendly interface.

It is worthy to be mentioned that the conception of an open design is not unique and was not first introduced in computer science. The same idea is widely exploited in many contemporary engineering sciences, such as architecture or mechanics. Though this approach bears other titles there, the four features listed above are typical for results of a professionally performed design.

1.3. Is open architecture desirable for a PSA simulating system?

Just as the open architecture concept is useful for most of the software, it also may be applied for the design of a PSA simulating system. The range of a PSA model complexity is rather wide. While in some cases the structure can be comprehended at one glance, other models may have a multilevel hierarchy of decomposition into submodels. Miscellaneous applications have significantly different sets of typical submodels. A PSA simulating system, which includes a fixed inextensible set (or library) of submodels has a limited use even if this set is rather versatile. Thus, the scalability and extensibility are highly desired features of a PSA simulating system.

The multitude of operating systems and hardware platforms and the absence of a leading platform imposes the necessity of support for more than one of them. The average lifetime of a PSA simulating system exceeds six years. The nowadays pace of hardware development speeds up the emergence of new operating systems and their versions. In six years the set of most widely used platforms alters significantly. The failure to easily port a PSA simulating system to newer platforms may cause its untimely disappearance because of disappearance of the single platform that it supports. The conclusion should be drawn that the portability of a PSA simulating

system is invaluable for achieving a longer lifetime and the acceptance of this system in the widest set of applications, which may be simulated with the help of PSA.

PSA modeling is often just a phase in a chain rather than entire process of data transformations. Input data for simulation may be resulting data of another system. And vise versa, the results of PSA simulation may be used somewhere else. A PSA system that is able to participate in such a chain should possess such feature of an open system as the interoperability.

The presence of a user friendly interface is almost the universal requirement for the contemporary software. In many applications this requirement seems to be redundant, especially when a program has only a small list of possible requests and responses. PSA is oriented to graphical representation of both simulating objects and rules. Not only a PSA simulating system must have a convenient visualization of objects and rules, but also it should supply a user with the means to visualize single and massive substitutions and collision occurrences. The experience of previously designed PSA simulating systems (such as CIM [5]) exploitation made it evident that debugging and observations of a PSA algorithm without an advanced visualization and user friendly interface is infernally complicated and time consuming. Thus all of the four features of open systems are either highly desirable or obligatory for a PSA simulating system.

1.4. Means of PSA simulating system open architecture implementation in WinALT

An attention was paid at the stage of WinALT design to meet all of the four features of open systems. To implement an extensible and scalable system WinALT was represented by the main binary executable file, which is constituted by the minimal set of modules required for WinALT execution, and an unlimited number of external modules.

The main binary executable file consists of a kernel and three main subsystems (Figure 1) [2]. These are the language, graphical user interface and visual design subsystems for PSA based models [1]. The WinALT kernel [6] is constituted by the object manager module and the external library support module, which is named ACL manager. As the name suggests the object manager implements operations with objects. Namely these are load, unload, create, delete, resize object; get, set cell value and name. ACL manager contains operations for external module support, such as load/unload. The WinALT external modules, which use the WinALT functions or some functions in other WinALT external modules are called ACLs. ACL stands for "Alt C Libraries". ACL is actually a dynamically linked library in Win32 [7, 8, 9] or a shared library in UNIX [10].

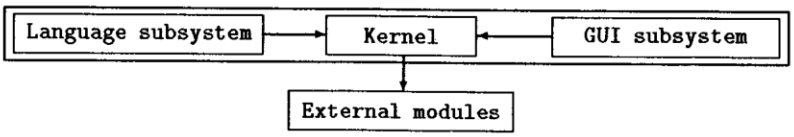

Figure 1. General structure of WinALT

Each external module that is intended to be used by WinALT must be registered. Whenever a user wants to exclude an external module from WinALT installation, the unregistration is activated. All the external module registration information is contained in a number of WinALT configuration objects. Usually these objects are located in the system directory.

All external modules provide WinALT with a standard set of interface functions. WinALT calls all the implemented operations of a certain module via these interface functions. On the other hand, WinALT provides external modules with an interface that allows the external modules to perform object transformations and other WinALT operations. The interface is implemented in the ACL manager [6].

The external module may be compiled and linked by any assembly or high level language compiler and linker or integrated developer's environment. The source of module interface functions is required. It may be accompanied by an unlimited number of other functions, source files. The WinALT library named `aclstd.lib` must be linked with each external module. A sample sequence of compilation and linkage stages for a C source texts is given in Figure 2.

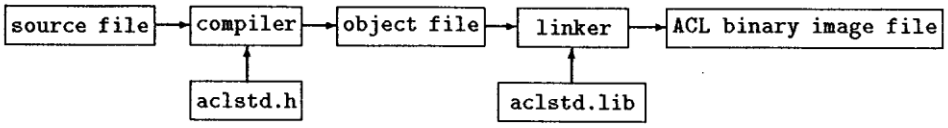

Figure 2. Generation of an ACL

Whenever a certain internal or external WinALT module has the need to utilize an operation that is implemented in another external module, ACL manager checks if the required module was already loaded. If it does not yet reside in memory, ACL manager loads it. Otherwise it increases the counter of its users. When the the module is not used any more, its user counter is decreased. When its value becomes zero, ACL manager unloads it.

WinALT was designed as a modular system. That allows to build simplified WinALT versions, e. g. versions without debug mode, graphical user's interface, etc. While such modules as graphical user's interface implementation considerably depend on a certain platform, the interpreter modules might be designed as portable. The WinALT language subsystem [6] uses only standard ANSI C [11] library and other WinALT interfaces and thus

can be easily ported to other platforms. ACL manager and object manager [6] constitute WinALT kernel, their implementation relies not only on ANSI C library, but also on some system functions, such as dynamically loaded libraries, memory mapped files and so on. But the porting of kernel also isn't a complicated task, as the system functions mentioned above are presented in virtually all contemporary operating systems. With the help of conditional compilation the single WinALT kernel source exists for all the platforms. The most platform dependent pieces of code were gathered in a single module, which is to be rewritten for each new platform. Currently WinALT implementation exists on Win32 and Linux platforms.

The interoperability of WinALT is maintained primarily in the object manager, as the main thing WinALT should share with other applications is the cellular object, which is the central WinALT data object. The object manager has the ability to adopt new file format for WinALT objects. The manager's design is examined in details in Section 4 of this article.

The last important feature of an open system is the user friendly interface. WinALT has an advanced graphical user's interface (GUI), which was described in [12]. The interface allows a user to view and edit objects and sources in a number of modes. In the debug mode a user gets a comprehensive set of tools to investigate a behaviour of an algorithm execution, to localize an error, to examine collision occurrences and so on. In Section 5 the description of the mean of WinALT GUI extensibility is given.

2. Interface for external modules

The set of implemented functions alters for different modules. Thus, a unified approach is required to retrieve these sets. Any ACL must have an exportable function named `ReqService`, which takes four parameters. The first parameter bears the number of the issued request. The rest of parameters are pointers, which have different meaning for different requests. For some requests these parameters are ignored. Having just a single function, which dispatch calls to other functions inside a module allows extending interfaces without a modification in the set of exportable functions. But on the other hand this approach implies overheads while making a function call. To improve the performance a few extensively used external functions are called directly without the `ReqService` dispatcher function. Retrieving and storing cell value functions are the best examples of such an exception. Besides the necessity of implementation of the dispatcher function, different types of external modules impose their own specific limitations in the interface.

In the passage above the interface, which allows calling external functions from inside WinALT has been discussed. But another way of interaction

between WinALT and external modules exists, when an external module calls a WinALT function. For example a module may ask WinALT to create, modify or delete certain objects. Some exotic modules may want to ask WinALT to terminate. The modules that depend on WinALT version need to retrieve the version number and so on. This type of interaction is implemented by WinALT language subsystem interface. The interface is reachable via `UA_SubmitServiceToACL` function. Just as `ReqService` it accepts a request number and three pointers. Pointer to this function is always sent to a module at the initialization stage and it should be kept unless a module does not have to handle WinALT objects or influence WinALT in other ways.

3. WinALT language extensibility

The same part of a model can often be used in more than one model. The means for its reusability would help to avoid its redundant reimplementation. It is well-known that coding is much less time consuming than testing and debugging. Each new implementation brings new errors. Instead of intensive exploitation of one well tested library a user has to debug a new one.

In WinALT a source code written in a certain high level language may be called from a simulating program. Such an import may be performed either for a source text or a compiled library. In both cases this source or binary file is linked with an interface library `aclstd.lib`. The linker creates an ACL library. Such a library may be loaded by a request in a WinALT program (Figure 3). The WinALT language gives two statements for library loading: `import` and `use`. Import makes interface functions visible by the long names with explicit specification of ACL name: `library::function`, for example, `math::sqrt`. Use statement enables short names, for example, `sqrt` for square root function in `math.acl`.

All the functions that should be visible in WinALT must be declared as interface functions. If a function is to be exported without modifications in its source (for example, a function from ANSI C standard library) a gate

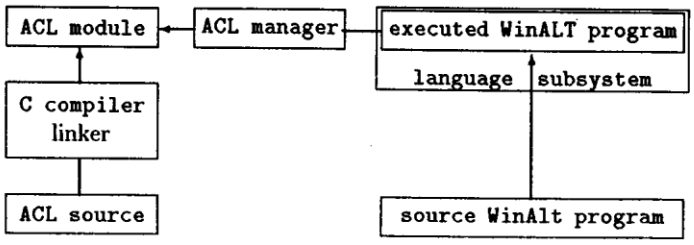

Figure 3. Calling an ACL function from a WinALT program

function should be implemented. The gate function is declared just like any other interface function. All it has to do is to call a real meaningful function. A gate function may be written manually or generated automatically from its prototype by **stg** utility. Prototypes for **stg** are similar to those in C with a few exceptions. They have the following syntax:

```
result_type function_name(type1 name1, type2 name2, ...,
                          typeN nameN);
```

All the functions that should be visible in WinALT must be declared as interface ones **function_name**, **name1**, **name2**, and **nameN** are valid C identifiers; **result_type**, **type1**, **type2**, and **typeN** are valid WinALT types (Table 1); **function_name** should be a name of an existing function. Formal parameter name may be omitted. The number of parameters has to coincide with that for the function named by **function_name**. ACL **xc** types are listed in Table 1. A list of examples of ANSI C and their respective **xc** prototypes is presented in Table 2. Comments are not allowed in **xc** files. For the sake of simplified implementation each prototype is placed on a single string. The last string contains '#' character.

Table 1. Relations between ACL, ANSI C, and Win32 types

xc type	C type	Win32 type	Description
`int`	`int`	`IN`	integer
`string`	`char*`	`LPSTR`	ASCII string
`boolean`	`int`	`BOOL`	logical
`float`	`float`	`FLOAT`	floating point value
`void`	—	—	error type
`char`	`char`	`CHAR`	ASCII character

Table 2. ANSI C and ACL prototypes

ANSI C prototype	xc prototype
`void func1(void);`	`void func1();`
`VOID func1(VOID);`	`void func1();`
`double sin(double);`	`float sin(float);`
`char* strcat(char*, char*);`	`string strcat(string, string);`

In many cases when only import to WinALT is required the **stg** utility would be sufficient. However, it is often necessary to handle WinALT objects from an ACL library. In such a case it is possible either to utilize WinALT interface directly with the help of macrodefinitions from aclstd.h header file or to use `aclstd.lib` functions.

```
ACL(ILF_MarkBorder1) /* the implementation */
{
  LPACL_OBJ objImage,
  /* pointer to an object with an image to work with */
            bufImage;
  /* keeps new pixel values until synchronization */
  LPSTR     tmpName;
  INT       xIdx, yIdx, xSize, ySize;

  ACL_ReturnInteger();
  /* notify that return value is integer */

  /* check if parameters passed have valid types */
  if((ACL_PARAM_TYPE(0)!=RT_TYPE_LPSTR)||(ACL_PARAM_TYPE(1)!=RT_TYPE_INT)||
    (ACL_PARAM_TYPE(2) != RT_TYPE_INT)) return ILF_ERR_INCORRECT_PARAM_TYPE;

  objImage = AS_StrToPtr(ACL_PARAM(0));
  /* get pointer to the object */
  if((objImage==NULL) || (objImage==(LPVOID)-1))
    return ILF_ERR_OBJ_NOT_FOUND;

  tmpName=AS_GetUniqueObjName();
   /* generate unique name for buffer object */
  ACL_GetObjSize(objImage, xSize, ySize);
   /* get size of the source object */
  bufImage = ACL_CreateObj(pszTmpObjName, xSize, ySize);
   /* create object  */

  for(yIdx=1;yIdx<ySize-2;yIdx++)for(xIdx=1;xIdx<xSize-2;xIdx++){
    INT value; BOOL bEqual;
    bEqual = TRUE;
    value = ACL_GetIntCellValue(objImage, xIdx, yIdx);
    if((value != ACL_GetIntCellValue(objImage, xIdx + 1, yIdx)) ||
       (value != ACL_GetIntCellValue(objImage, xIdx, yIdx + 1)) ||
       (value != ACL_GetIntCellValue(objImage, xIdx - 1, yIdx)) ||
       (value != ACL_GetIntCellValue(objImage, xIdx, yIdx-1)))
         bEqual = FALSE;

    /*set ACL_PARAM(2) value if the point belongs to border,
    ACL_PARAM(1) otherwise */
    ACL_SetIntCellValue(bufImage, bEqual ? ACL_PARAM(1) : ACL_PARAM(2),
                        xIdx,yIdx);
}
/* local function that copies all cells from bufImage to objImage */
SyncChanges(objImage, objNewImage); return ILF_OK;
```

Figure 4. An ACL function implementation sample

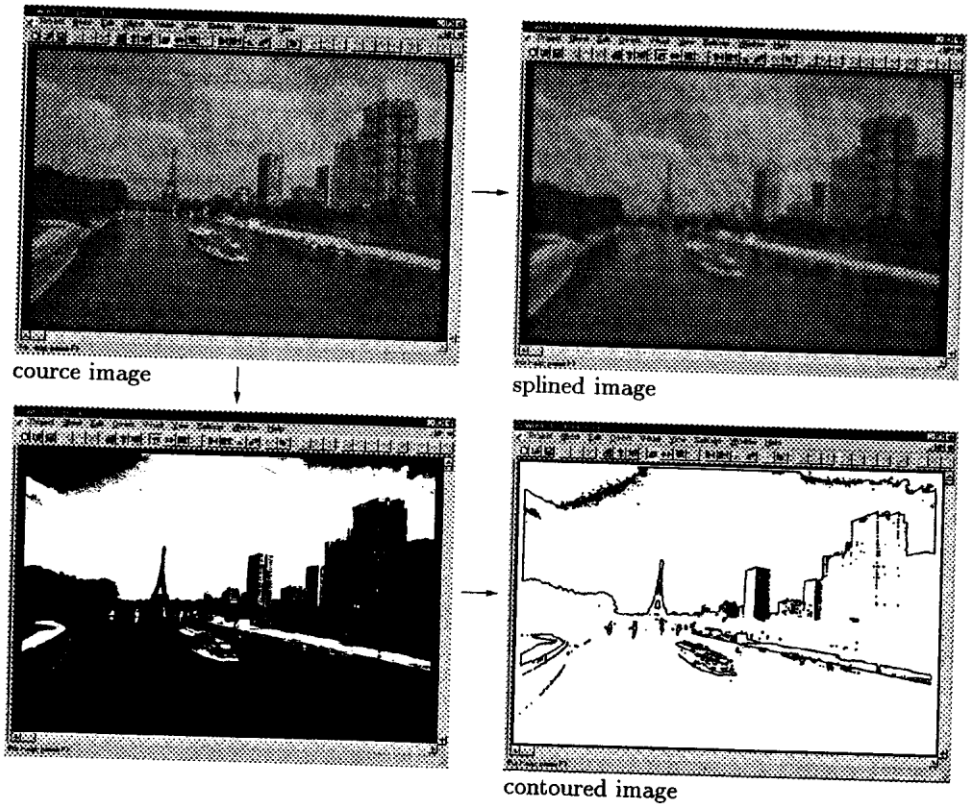

Figure 5. The source, contoured and splined images

A sample of an ACL function, which marks a contour of a 2D image, is depicted in Figure 4. A pixel (a cell) is assumed as one of a contour if at least one of its neighbor cell values alters from that of the pixel. The source, intermediate and contoured images are shown in Figure 5.

4. Custom object file format support

Nowadays a lot of miscellaneous file formats for different types of information exist. Some of these specifications, such as GIF, are revised periodically. Spontaneously or because of convenience different applications have different standard formats. A WinALT user would not feel comfortable if he is forced to utilize a fixed set of formats without the ability to add a new one.

WinALT object manager [6] supports multiple file formats. Each file format is represented be a so called object driver, which is a dynamically linked library with a specific set of exported functions. WinALT package contains a number of such drivers. Some of them support internal object formats. Others implement some widely spread file formats, such as BMP.

A user may implement his own object drivers to enable the usage of his custom file format. Object driver's registration may be performed either by WinALT object driver registration dialogue or by modification and execution of `config.src` that is located in WinALT BIN directory. After the registration WinALT configuration contains the information about the new object driver (the driver's path, object prefix, file extension). When object manager encounters an object containing a prefix or file name extension relevant to this object driver, it loads the driver if it was not loaded before. When no more objects of a certain type reside in memory, their driver is unloaded by the object manager. Each object driver has a number of obligatory functions. These are object load and unload, cell value and name read and write, get object dimensions, amount of memory required to keep an object with specified dimensions.

Currently the following set of object drivers is implemented in WinALT:

1) the default driver;
2) the driver for objects with 1, 2 and 4 byte integer cells;
3) the driver for 1 bit logical cell values (suits for classical cellular object simulating [13]);
4) a number of object drivers for some subformats of BMP format.

5. WinALT graphical user's interface extensibility

5.1. General description

PSA is applicable for a versatile set of real problems. WinALT as an PSA based system should cover most of them. But the difference does not entirely reside only in the set of typical submodels or the most widely accepted file formats to keep objects for a certain problem. Different tasks require as well their own way of graphical representation of source, intermediate or resulting objects.

The best form of visualization comes from tight connection with real physical process in a model. For example, the results of sound or seismic signal filtering have one dimension and relatively large set of discrete values. Thus, the best way to represent such type of data is to show the dependence of signal level from time. The location on X axis denotes the position or time, while that on Y axis shows the value of signal. Colour may be used to place a number of such signals (or objects in PSA terminology) into the same region on the screen. The visualization of images evidently should have two dimensions with colour of each point denoting its value. PSA models for digital device simulation may require 2D or 3D visualization. Depending

on a cell complexity, a value may be showed as a colour square or a certain text in a rectangular area.

Other PSA applications may reveal more demands and peculiarities on how to show a cellular object. Thus, the ability to include and use new modes of object visualization is an essential feature of an PSA based system because of its influence on integral fitness for a certain application.

In WinALT, the custom object visualization mode support is implemented in the module named Object Visualization Engine.

5.2. Structure of the object visualization engine

Object visualization engine (OVE) is the part of WinALT GUI subsystem. It is built above the WinALT object manager. This module also uses extensively Win32 API GUI functions, because its main purpose is visualization. OVE is activated by other modules of WinALT GUI subsystem and by the language subsystem.

The principal data structure of the module is the visual object. It includes object manager logical object as its part. Other parts of this structure reflect miscellaneous parameters of visualization and object editing. For example, there are such fields as location and size on the screen, "undo" buffer to cancel recent modifications of an object, the visibility of object name, of the ruler (the rectangular region at the top and left edges of a visual object, its only destination is to show the coordinates of X and Y axis and so on.

5.3. Modes of visualization and external mode support libraries

Each visual object is shown in a certain mode of visualization. One such mode is supported by an ACL named OVD. OVD stands for Object Visualization Driver. It uses the same interfaces as any other ACL library. The painted image in a window is the result of cooperative work of the OVE and one or several OVD libraries. A window can contain one visual object in viewer or edit mode or a collection of objects in viewer mode. Each object is located in a rectangular region, which is divided into client and non-client areas. The former is painted by OVD, while the latter is created by OVE and includes title, frame and ruler and edit string at the bottom. The title is used only for multiple objects per window.

OVD may have a number of optional exportable functions, but there is only one obligatory. It has the symbolic name `OVD_Paint` and it is responsible for painting the client area of a visual object specified as a parameter in a window. It is the responsibility of OVD not to go beyond the borders of the client area. The source text in Figure 6 shows a simplified version of `OVD_Paint`, which paints integer cell values for a 2D visual object or a

```
_declspec(dllexport) VOID OVD_Paint(PVISOBJ pvo, PDRAWINFO hdc)
{
    INT     xSize, ySize, zSize, xMax, yMax, xCurCoord, yCurCoord;
    LPACL_CELL  pcell;
    struct Pos  pos;
    CHAR        buf[8];

    ACL_GetObjSize(pvo->plo, &xSize, &ySize, &zSize);
    pos.z = pvo->posScroll.z;
    xMax = min(xSize, pvo->sizeWin.x / pvo->sizeUnit.x + pvo->posScroll.x);
    yMax = min(ySize, pvo->sizeWin.y / pvo->sizeUnit.y + pvo->posScroll.y);

    for(pos.y = pvo->posScroll.y, yCurCoord = pvo->posWin.y; pos.y < yMax;
        pos.y++, yCurCoord += Y_CELL){
    for(pos.x = pvo->posScroll.x, xCurCoord = pvo->posWin.x; pos.x < xMax;
        pos.x++, xCurCoord += pvo->sizeUnit.x)
    {
        sprintf(buf, "%d", pcell->value);
        TextOut(hdc, xCurCoord, yCurCoord, buf, strlen(buf));
    }
}
```

Figure 6. `OVD_Paint` implementation sample

```
_declspec(dllexport) LPVOID OVD_Request(PVISOBJ pvo, INT req, LPVOID p1,
                                        LPVOID p2)
{
    struct Pos  pos;

    switch(req)
    {
    ...
    case OVD_REQ_CORRECT_SIZES:
        ACL_GetObjSize(pvo->plo, &pos.x, &pos.y, &pos.z);
        if(pvo->sizeWin.x < X_CELL) pvo->sizeWin.x = X_CELL;
        if(pvo->sizeWin.y < Y_CELL) pvo->sizeWin.y = Y_CELL;
        pvo->sizeUnit.x = pvo->sizeWin.x / pos.x;
        if(pvo->sizeUnit.x < X_CELL)pvo->sizeUnit.x = X_CELL;
        else    pvo->sizeWin.x = pvo->sizeUnit.x * pos.x;
        if(pvo->sizeWin.y > (pvo->sizeUnit.y * pos.y))
            pvo->sizeWin.y = pvo->sizeUnit.y * pos.y;
        return (LPVOID)1;
    ...
    }
}
```

Figure 7. `OVD_Request` implementation sample

layer in 2D object. `xCurCoord` and `yCurCoord` denote logical coordinates in window. pos is a structure that contains the position of a cell in object.

`OVD_Request` is an optional OVD function, which allows to adjust OVE settings. For example, by default cell scale vary from 1 to 1024 pixels. But for some modes it senseless to decrease the scale below a certain value. Or the standard ruler is useless for some modes. Modifications of scale, size, altering the visibility of vertical or horizontal ruler may be implemented in `OVD_Request`. The source test presented in Figure 7 demonstrates a partial implementation of `OVD_Request`, which corrects sizes of a visual object. This sample has fixed minimal sizes for X axis (`X_CELL`) and Y axis (`Y_CELL`). It limits the maximal height, but the width is unlimited, for each increment of width, it increases X axis cell size.

5.4. Other means of graphical user's interface extensibility

WinALT GUI subsystem has an API that allows to create and manage child windows in the main WinALT window. This API is available for external modules via ACL interface. Thus, a WinALT program with a help of a certain ACL may perform any Win32 GUI operations. Should anything that goes beyond limits of rectangular cellular objects or the conception of visual modes arise, it would be implemented with the help of this API. This API is also supported for simplified WinALT versions under Win32.

References

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

[2] S.V. Piskunov, *WinALT - a simulation system for computations with spacial parallelism*, Bull. Nov. Comp. Center, Comp. Science, No. 6, 1997, 71–85.

[3] N. Wirth, *A plea for lean software*, IEEE Computer, **28**, No. 2, 1995, 64–68.

[4] E. Filinov, *The selection and development of the open system environment conceptual model*, Open Systems, **6**, No. 14, 1995, 32–46.

[5] Yu. Pogudin, *ALT - a graphical system for parallel microprogramming*, Parallel Algorithm and Structures and Structures, Computer Center, Novosibirsk, 1991, 77–88 (in Russian).

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

[7] Charles Petzold, *Programming Windows 95*, Microsoft Press, 1996.

[8] Jeffrey Richter, *Advanced Windows*, Microsoft Press, 1995.

[9] David J. Kruglinski, *Inside Visual C++*, Microsoft Press, 1996.

[10] *LINUX manual pages*, LINUX Redhat 5.0, 1997.

[11] *Turbo C++, second edition. User's guide*, Borland Intenational, Scotts Valley, 1991.

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

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