Abstract. *Caper* is a parallel programming language, which supports declarative parallel computations and control of all architectures by Flynn. Caper based on virtual machines system, including own parallel virtual machine. Caper has a self-organization and asynchronous events processing programming means. Represented language has various variables with different scope, time of creation and survival time. Besides, Caper has so called “controlled variables” or variables with statuses, which allow regulate usage of variables by different parallel processes. This property allows to create programs not depended on multitasking management of operating systems.

1. Introduction in Principles of Parallel Programming Language Caper

Caper is a parallel programming language based on the following principles:

- the possibility of calculations in the terms of the main parallel models;
- parallel performing the program structure components without using the parallelizing means of operating systems;
- the possibility of program self-organization during a computing process, dynamic compiling and performing a source code and individual commands, dynamic composing the running code by means of loading and removing object modules;
- controlling the computing process based on different classes events;
- the object-oriented programming for parallel calculations.

The represented version of Caper is the fourth and was developed on basis of investigations, which were started in 1985 and were presented in different publications (see below). Caper’s compiler, virtual machine and some others means for programs development and debugging are developed now.

Caper is based on the virtual machine (CVM), which supports both sequential and parallel programs performing. *Caper* instructions provide synchronous and asynchronous starts of program procedures with using *Caper’s* own model of pseudo-parallelism, called as "command-by-command" and the familiar model with time quanta. *Caper* allows start multiple parallel procedures with common data, a single procedure with multiple data, multiple parallel procedures with multiple data, in other words all architectural schemes by Flynn classification. Caper has means for mass parallel starts of parallel processes by special interpretation of procedural prototypes.

Caper is portable. It is independent from calculation model (sequential or parallel), which is used on the computing set. From the viewpoint of an operating system a program in *Caper* can be considered as a single task without subtasks.

Caper allows dynamically change single command or fragments of a running program, or remove separated object modules and load others object modules, compile a source code and perform it.

All enumerated possibilities of the language machine and self-organization means allow to transport source codes, object modules and fragments of the performed code with data to different computers united by network or any other means.

Caper provides possibility to describe various events and to assign the procedures of asynchronous sequential or parallel processing them. The language machine distinguishes classes and subclasses of events. The language is provided with multiple facilities of events managing, e.g. defining events, freezing, de-freezing, removing and initializing them.

All *Caper*'s variables can be typified or polymorphic and differ in the scope, creating way and existing time. Besides usual variables *Caper* has such resource as managing variables, which aimed for concurrent solutions. These variables are characterized by the states, which can be set or changed at the different moments of calculations to regulate access to the variables.

The language is provided with special means, which make it easy to develop re-enterable procedures: any program module is either re-enterable or not re-enterable according to the way variables are described in.

The principles, on which based *Caper* allow say about following ways of evaluations. In first, we project different real-parallel Virtual Machines development for supporting execution of Caper’s programs into multiprocessor computers and multi-machines complexes (including Clusters). The conception of Caper demands reviewing of role of operating systems: we consider operating systems as means, which will support Virtual Machines interactions.

We represent in this text the main ideas and statements of Caper (Caper is supported by some others secondary constructions and has more then 400 functions of Virtual Machine, different functionalities which are supporting by object modules in Caper). We investigated and developed a few methods of programming by different styles in Caper.

We have large experience (more then 10 years for different versions) of programming in Caper in following areas: images processing, information searching tasks, solution searching tasks and others. During execution of some tasks more then 100 thousands parallel processes were started and executed in a single processor (Pentium II, 512 Mb RAM). Caper have own means for parallel reading and processing files, parallel searching in memory and in databases, etc. Real parallel execution of Caper's program was approved for image processing: source and object modules of Caper are transportable and were migrated to different computers of network for realization of own tasks.

Caper will be represented here by short descriptions.

2. Alphanumerical System, Program Strings

The *Caper* is founded on ASCII character set. We differentiate logical and physical strings. A physical string is the string of a text editor. A logical string is a string which occupies a few physical strings and defined by sign '=>' - the sign of string continuation. All signs placed in a physical string after '=>' are ignored. A physical string can be divided by ';' into a few logical strings.

Caper has signs for comments definition: '*' and '//' for a logical comment string, '/*' and '*/' - comment brackets, and some other variants.

3. Operators, Commands and Expressions

The command in Caper is the list of arithmetic or logical expressions, or a control operator. The expression is similar to expression definition in a many others programming languages, in particular, in C.

The list of expressions is

<expression 1> [, < expression 2>] . . . [, < expression N>]

which called as COMMAND. The command may be empty, if logical string don't contains the list of expressions or control statement.

The set of operators contains the following:

Assignment operators:
:= - assignment operator;
= - assignment by reference operator;

Arithmetic operators:
+ - summation;
- - subtraction;
* - multiplication;
** - exponentiation;
/ - division;
% - division by mod;
+= - self-summation;
-= - self-subtraction;
*= - self-multiplication;
/= - self-division;
%= - self-division by mod;

Logical and comparison operators:
== - equal;
!= - not equal;
> - greater then;
< - less then;
>= - greater or equal;
<= - less or equal;
&& - logical AND;
|| - logical OR;
! - logical NOT.

Binary operators:

| - OR;
& - AND;
>> - shift to right;
<< - shift to left;
|= - self-OR;
&= - self-AND;

Unary pointing operators:
& - pointing of block or global label by variable’s content;
@ - taking of a reference to the variable or place;
$ - taking of place status.

=* - the copying of compound types of data.

All operators have a weight in expression; all self-changing operators have the assignment operators weight.

Examples:
The list of expressions is represented below
```
var1 := ( var2 := arr1[i, j, k-i, k-j] + arr2[n] ) =>
            * 10.2'D ,        =>
 i := ( var1 + var2) /   =>
       (j := (var1 * var2)), n := var1 + =>
       ( var2 *= n ) + k
```

4. Program Structure

A program in *Caper* is an aggregate of so called blocks. Blocks have different types: blocks of commands, data, text, image, array and string. Blocks of commands are general logical executable units of programs in *Caper*. Any command is either a sequence of arithmetical or logical expressions or a control statement. *Caper* expressions are constructed from arithmetic, comparing, logical, binary, assignment and some other operators. All commands are placed in logical strings. In fact, any block can be interpreted as array of elements.
A block is a general aggregate of executable commands and different static data.

```
BLOCK <blockname> [STATIC] [ (<fparm 1>, . . . , <fparm N>)]
                           [AS <blocktype> ]
    . . .
ENDBLOCK
```

<blockname> is an identifier.
<fparm 1> - <fparm N> are block formal parameters.
<blocktype> - COMMAND for a commands block.
DATA - for a data block (the set of literal values).
TEXT - for an aggregate of text strings.
IMAGE - for arbitrary bytes sequence.
ARRAY - for static arrays.

If <blocktype> is not stated, it is fixed as COMMAND. The block with parameters must be defined for COMMAND type only.
STATIC keyword can be used only with COMMAND type. This means that places for these parameters will be reserved into body of compiled module. During call of this block parameters values will be put into reserved places.
If STATIC is absent, then all parameters during calls will be created dynamically.
Any file with source code of Caper will be interpreted by Caper's compiler as block of commands. The Caper's compiler assigns the MAIN default name to created object module, excluding cases, when the name of module is assigned directly by compilation command.
If a block has IMAGE type

BLOCK <blockname> AS IMAGE [OF <filename>]

then the block body will be loaded from the file <filename> in a compilation step.
Block with type DATA contains constant numeric or string literals.
Block with TEXT contains strings, which separates by the line folding signs.
Static arrays can be created by

BLOCK <blockname> AS ARRAY(<type>, <initial value>, <dimention>)

Every block of any type must be bounded by ENDBLOCK keyword.

Blocks of all types can be included in any COMMAND block.
Block elements are pointed as array elements: <blockname> '[' <index>']'.
Caper supports functions (procedures) of two types: machine (or environment) functions and programmed functions, which are defined by programmer. The machine functions are intended for CVM call.

FUNC <block name> [STATIC] [(<fparm 1>, <fparm 2>, ..., <fparm N>)]
. . .
ENDFUNC

and

FLICK <block name> [STATIC] [(<fparm 1>, <fparm 2>, ..., <fparm N>)]
. . .
ENDFLICK

The result of source code compilation is the *Caper*'s object module or an executable file. *Caper* compiler creates CVM byte-code in different regimes, which can be selected by *Caper*'s compiler commands. In particular, compiler can create so called "critical fragments" – program fragments, which monopolize resources of CVM. FUNC locks parallel machine of Caper, FLICK locks parallel machine and, additionally, changes CVM regime on SOLE – blocking events submachine of CVM. Any return from flick changes CVM regime on that, in which worked the calling block.

All functions are compiled as critical fragments. All flicks are functions, which locks all events from operating system to CVM.

Example:

```
// module's body is block of commands
x := 1
y := 1
 . . .
block bl_name1
  . . .
  block bl_name2 static ( parm1, parm2 )
    . . .

    flick fl_name1 static ( fparm1, fparm2 )
    . . .
    endflick

    func f_name1 ( fparm1, fparm2 )
    . . .
    endfunc
  endblock

  block  bl_name3

    func f_name31 ( fparm1, fparm2 )
    . . .
    endfunc

    block bl_name4 ( bparm )
     . . .
      block bl_name5 ( bparm )
      . . .
        flick fl_name31 ( fparm1, fparm2 )
        . . .
        endflick
      endblock
      . . .
  endblock

endblock
```

5. Variables and Places

Variables in *Caper* are polymorphic or can be typified. *Caper* has also so called "controlled variables", or *Places*. They are global variables with status. All polymorphic variables are internally typified and do not demand from their types controlling - *Caper* machine does it. Any variable or *place* can be undefined – NULL type.

Compiler of Caper realizes checking of types and types converting. Caper's structural data generate new types. And more – declaration of new Caper's command block of any style generates new procedural type. After such declaration we can describe new blocks by this type.

Caper doesn't permit direct access to computer memory: all structural data are referenced by variables.

The strings of *Caper* are interpreted as integral data and as arrays of bytes (symbols), which can be addressed as arrays elements.

Places of *Caper* have a special meaning. Every place-variable has one of the following states: WRITE_ONLY, READ_ONLY, LOCKED, FREE or UNLOCKED (synonyms). Usage of *Places* is controlled by CVM. At the first moment all defined *places* have FREE status. A *place* owner is the block, which is the first to set a not FREE status. Any attempts to use a *place* or its value from the others blocks (which are not the *place* owners) and conflicts with the *place* status will cause an internal CVM error.

The control of place is the control of resource which this place is represents.

All *Caper* variables are different in their scope, creating way and existence time. There are Public (global), Private and Local variables. The statements of variables definition are formed by

{ PUBLIC | PLACES | PRIVATE | LOCAL } <variables list with initialization>

<variables list with initialization> ::= [<type_array descriptor>] <variable name > [:= <expression>]
{[<type_array descriptor>] <variable name > [:= <expression>] }
<type_array descriptor> ::= <type descriptor> [<array descriptor>]
<type descriptor> ::= '**<**' [<aggregate>] <type> '**>**'
<aggregate> ::= **`size | memnum | array | as | implant`**
<type> ::= <basic type> | <generated type>
<basic type> ::= **null** | **double** | **float** | **great** | **long** | **word** | **int** | **half** | **shalf** |
byte | **char** | **addr** | **string** | **array** | **var** | **place** | **collect** | **block**
<generated type> is identifier and one of tag names of structures or prototypes (see below).
<array descriptor> ::= '[' [<list of dimensions>] ']'

Examples:
private <**int**> var := 0, <**float**> [] arrOfFloat
local <**byte**> char := 0, <**double**> [10,20] arrOfDouble

Aggregates in type definitions of variables have different applications: different groups of aggregates can be use only in special cases and combinations. So, **size** and **memnum** can be use only in arithmetical expressions. These aggregates form numeric values by <type descriptor>: **size** forms the size of type in bytes, **memnum** forms the number of members of structural types.

var := < **size int**> + 10 ;* the value of expression is 14
var := <**memnum** tagSomStructure> ** 2 ;* if tagSomStructure has 4 members, then var contains 16

Others aggregates will be described later.

The type of variable has elongated effect in any definition: if some variable hasn't type definition, then previous variable's type will be assigned for current variable. In

public <**block**> bl_var1, bl_var2, <**string**> str1, str2, str3

bl_var1, bl_var2 have the same type **block** (references to blocks), str1, str2, str3 are references to strings.

If a variable hasn't got initialization, then compiler sets the **null** value (undefined) to this variable.

Public variables and *places* are created in internal memory of CVM and can be defined and deleted in any place of a program. They can be redefined.

Private variables are visible only in the block and its sub-blocks. If private variables were defined outside of module blocks then their scope is the whole program module. Private variables are static. They are placed in object module's body. These variables are being deleted with the module's body (see the modules removing means).

Local variables' scope is only a block body without sub-blocks. These variables are created after executing LOCAL statement and deleted after the block termination.

Block parameters have the same meaning as local variables. They are created at the moment of the block call and deleted after the block termination. If STATIC was defined, then parameters pool will not be created dynamically, places for parameters will be created by compiler in the module's body.

If public-, private-, define-variables are leaded by specificator INITIAL then repetitive initialization of variables will be prohibited.

LockF(<place name>, <status>) assigns the status of a *place*. <status> is one of above enumerated statuses. *Caper* allows define a set of user statuses of *places* in a program.

Public variable and *places* can be deleted by DELETE statement with the list of variable or *place* names.

Examples:

```
private  <int> pr_var1 := 0, pr_var2, =>
         <var> [100] pr_var3   ;* pr_var3 is the reference to array of variables

block  bl_name1  static ( parm1, parm2 )  ;* parm1 and parm2 are visible only in block, but not in
                                          ;* sub-blocks
   pr_var2 += parm2           ;* execution error will be occurred: pr_var2 has null value
   . . .
   local <int> var1 := parm1+10, var2 := parm2 +20
         var3 := var1 + var2
   . . .
   pr_var1 += var1            ;* pr_var1 is visible here
endblock

flick SomeFlick
  . . .
  pr_var2 := var2 + 10    ;* compiler error will be initiated: var2 isn't visible here
  . . .
endflick
```

6. Block Names Visibility and Prototypes (first definitions).

Block names are different in their scope. Such, all block can be public and visible in all program or internal for module. Internal names are representing by the following statements:

<u>INTERNAL</u> <block prototype 1>, < block prototype 2>, …,
<block prototype N>

where

<block prototype> ::= [<returned value descriptor>] <block name>
[([<parameters descriptions list>])]
<parameters descriptions list> :: = [<type_array descriptor>] <variable name >{, [<type_array descriptor>] <variable name>}
<returned value descriptor> :: = <type_array descriptor>

The internal blocks aren't visible from others modules and can't be called from other modules directly by name. Every defined block creates procedural data type equal to block name.

```
internal <great> [10] SomeBlock ( <byte> [] bytesArray, <int> x, y ),  =>
              <null> SomeOtherBlock()
```

SomeBlock and SomeOtherBlock will be fixed as new types (procedural types).

```
private <SomeBlock> refToBlock1,  <SomeOtherBlock> refToBlock2
```

defines variables, which can be used as references to real blocks SomeBlock and SomeOtherBlock.
We can pass such variables and realize calls

```
refToBlock1( someArrayOfBytes, 10, 20 )
refToBlock2()
```

<u>PROTOTYPE</u> <block prototype 1>, < block prototype 2>, … ,
<block prototype N>

defines blocks descriptions class, but not real blocks. <block name> in INTERNAL demands real block definition, but <block name> must be used for only indirect definitions of real blocks. For indirect definitions in INTERNAL must be used AS aggregator:

prototype <**great**> [10] TypeOfBlock (<**byte**> [] bytesArray, <**int**> x, y)
internal <as TypeOfBlock**>** SomeBlock1, SomeBlock2

SomeBlock1, SomeBlock2 are names of real blocks.

Prototypes in *Caper* have dual interpretation. In first, they are procedural class definitions and in second they are definitions of special structural data types. The possibilities to manipulate with them as with data we'll consider later.

7. Constants and string literals.

Constant literals in Caper are represented by the following:

Type	**Variants**
Doubles	- 123.45'D, -0.12345e3'D, 12345E-2'D
Floats	- 123.45, -0.12345e3, 12345E-2
Greats	- 1234567'G, 0'G, 10'G
Longs	- -1234567'L, 0'L, -10'L
Words	- 12345'W, 0'W
Integers	- 123, -12345'I, -1'I, 10'I
Half words	- 12345'H, 1'H, 10'H
Signed Half words	- -123'J, -1'J, 10'J
Bytes	- 123'B, 200'B, 255'B, 'A', 'a', 'B', 'b', '+', '-'
Characters	- 123'C, 127'C, -127'C

The symbol " ' " (quotation mark), which follows for numeric part of literal, delimits descriptor of type. This descriptor prescribes to compiler create literal value of defined type.
If descriptor is omitted, then compiler sets by default type. If dot sign is absent and/or exponent sign ('e' or 'E') is absent then compiler converts numeric string to integer ('I') type. In other case, compiler converts to float type.
String literals and symbolic literals are defined by double quotation marks and by pair of single quotation marks correspondingly:
string literal –
"Hello!", "It's Caper", "A"
symbolic literal –
'A' , '1', '#'

Besides, we can represent binary and hexadecimal numbers:
binary -
0b00000100 - number 4;
0b00000011 - number 3;
0b00000010 - number 2;
(all these numbers will be represented by type integer 'I')
0b0000000100000011'H - the number 300 will be converted into 2 bytes (half integer);
0b00000011'B - the number 3 will be converted into 1 byte and will have 'B' (byte) type.

hexadecimals -
0x01'B - the number 1 will be converted into 1 byte (type 'B);
0x0A'D - 10 will be converted into double float (8 bytes);

0x000001F2 - 498 has type integer.

```
var1 := 0x01
var2 := 12345  (i.e. 12345'I )
var1 := 0           ;* integer 0
var2 := 10.2e3      ;* float
var3 := 10'F        ;* float
var4 := 1.252       ;* float
var5 := 3'B         ;* byte
var6 := 'A'         ;* character as byte
var7 := "Hi!"       ;* literal string
```

8. Structures and Collections

Caper allows describe and create structures and collections of variables:

<u>STRUCT</u> <tag_name> '{' <member descriptor> { , <member descriptor> } '}'
['{' [<constructor>] [, <destructor>] '}']

<member descriptor> ::= <type_array descriptor> <variable name > [<ref_build_flag>] |
<implantation>
<tag_name> ::= <identifier>
<ref_build_flag> ::= **build** | **ref**
< implantation > ::= '<' **implant** <tag_nameX> '>'

<ref_build_flag> defines insertion of body of described member into current structure. BUILD keyword demands insertion, REF keyword (or omitting of these keywords) signify that this variable is reference to the compound data.
<tag_nameX> is identifier and must be not equal to <tag_name>.
<constructor> and <destructor> are identifiers and names of real blocks.
About real building of structures and others compound data we describes later (see **build** and **build_by** statements descriptions)

struct tagSomeStru { <**int**> iVar, =>
<**float**> [] arrOfFloat, =>
<**byte**> [100] arrOfByte **build,** =>
<**as** TypeOfBlock> myBlock }

It is obvious, that using in definition

<**float**> [] arrOfFloat **build**

is incorrect, because the size of array isn't defined. Compiler of Caper informs about error.
<u>BUILD</u> and <u>REF</u> keywords are meaningless for simple (basic) types.

struct tagOtherStru { <**int>** [] arrOfInt, =>
<**implant** tagSomeStru>**,** =>
<tagSomeStru> stru1 **ref,** =>
<tagSomeStru> stru2 **build** =>
}

where <**implant** tagSomeStru> demands to include in the defined structure the all members of tagSomeStru structure, but stru2 refer to structure, which will be built-in here. stru1 member is reference to structure tagSomeStru, which can be created outside. Let struVar variable contains the reference to tagOtherStru structure. Then we can use members:

```
struVar.arrOfInt      := otherArray
struVar.arrOfInt[1] := 10
struVar.iVar          := 10         ;* this member was implanted
struVar.arrOfByte[50] := ‘A’  ;* this member was implanted
struVar.myBlock(struVar.arrOfByte, struVar.iVar, struVar.arrOfInt[1] )
struVar.stru1.iVar := 20  ;* will be terminated on execution error, if some built structure of
                                ;* tagSomeStru type will not be assigned to struVar.stru1
struVar.stru2.iVar := 30  ;* will work correctly, because the body of struVar.stru2 exists
struVar.stru2.myBlock( struVar.stru2. arrOfByte, struVar.iVar, struVar.arrOfInt[1] )
```

Collection is a polymorphic variables group, which must be described and can be created dynamically or statically in a module body.

```
COLLECTION <tag_name> { <varyable name 1>,
                        <varyable name 2>,
                         ... ,
                        <varyable name N>
                      } [ ‘{‘ [<constructor>] [, <destructor> ] ‘}’ ]
```

<tag_name> - identifier, which represents collection;
<varyable name> - identifier and collection’s internal variable name.
<constructor> and <destructor> are identifiers and names of real blocks.

The forms of referencing of collection’s members are following:

<variable name>.<expression> | <variable name>.<literal string>

where <expression> - arithmetical expression, <variable name> contains reference to collection,

Literal string must define the name of member of collection, but arithmetical expression defines the number of member in collection. Collection’s member selection will be supported every time by collection descriptor.

Both collections and structures can be dynamically created by

<variable name> := <tag_name> ‘{‘ [<list of expression>] ‘}’

For static case:

static <tag_name> ‘{‘ [<list of expression>] ‘}’ <list of variables name>

Examples:
collection SomeCollection { var1, var2, var3 }
collection OtherCollection { var1, mem1, mem3 }

private <SomeCollection**>** myColl1, myColl2

myColl1 := SomeCollection { 10, “ABC”, ‘C’ }
myColl2 := SomeCollection { 20, “CDE”, ‘A’ }

myColl1.”var1” + myColl2.”var1” ;* equal to 30
myColl1.”var3” == (myColl1.var2)[1] ;* these elements are equal
myColl1.1 is the same that myColl1.”var1”,

myColl1.2 is the same that myColl1."var2",
myColl1.3 is the same that myColl1."var3",

Static variant of creation (in the body of compiled module) and assignment of reference to collection to two variables:

static SomeCollection { 10, "ABC", 'A' } myColl1, myColl2

Collections descriptions are stored into modules body, in which such collections were defined. Correspondingly, these descriptions are using during program execution.
Collections are interesting by methods of usage. In first, we can use them for organization of different hidden interfaces between program modules:

9. Compound Data Building and Destroying

The following statements support building of compound data.

<u>BUILD</u> <variable building descriptor> {, <variable building descriptor> }

<variable building descriptor> ::= <variable name>[(< constructor parameters list >)]
[: <block call descriptor> { : <block call descriptor> . . . }]

<variable name> here must represent only structure, array or collection.
<parameters list> is ordinary list of parameters, which will be passed to constructor.
<block call descriptor> ::= <block name>([<parameters list>])
: <block call descriptor> means, that after building of body of compound data and constructor calling (for structures and collections) Caper's Virtual Machine will call block with <block name> and parameters. These means allows enhance of construction procedure by usage of additional procedures.
Usage in the BUILD of simple variables (variables of basic types) is incorrect and will cause error message from compiler.

*Example*s:

internal <**null**> Construct(<**int**> x, <**float**> z), <null> Destruct(<**int**> y), =>
<**null**> AddConstructor(<**int**> x)
struct tagSomeStru { <**int**> iVar, <**float**> [] arrOfFloat, =>
<**byte**> [100] arrOfByte **build,** =>
<**as** TypeOfBlock> myBlock } => TypeOfBlock defined above
{ Construct, Destruct }

private <**int**> [] arrOfInt, <**float**> [10,20] arrOfFloat, =>
<tagSomeStru> stru

build arrOfFloat, stru(10, -0.5) : AddConstructor(100)

This example shows separable declaration and building of data: we declare of variables (**private** statement) and later we build them by the **build** statement.
We can't use

build arrOfInt   ;* incorrect usage

because sizes of array are undefined. Correct form:

build arrOfInt(10, 20, 30)   - parameters describes dimension and sizes of array

For building of arrays of structures with undefined dimension and sizes we can use other form:

BUILD <array’s variable name> [(<array dimensions>)] [(<constructor parameters list>)]
[: <block call descriptor> { : <block call descriptor> . . . }]

Example:
private <tagSomeStru> [] arrOfStru
. . .
build arrOfStru(10,10)(10, -0.5) : AddConstructor(100)

The following statement allows destroy of compound data:

DELETE <variable destroying descriptor> {, <variable destroying descriptor> }

<variable destroying descriptor> ::= <variable name>[(< destructor parameters list >)]
[: <block call descriptor> { : <block call descriptor> . . . }]
< destructor parameters list > is the list of parameters for destructor.

Example:
delete stru(100) ;* 100 is parameter for destructor

BUILD creates body of data, calls constructor and later additional constructors.
DELETE calls destructor, additional destructors and then removes body of data from memory.

Other method for creating and deleting is supported by statements **build_by** and **delete_by**, which

BUILD_BY <array’s variable name> [(<array dimensions>)] [(<constructor parameters list>)]
[: <block call descriptor> { : <block call descriptor> . . . }]
BUILD_BY <array’s variable name> [(<array dimensions>)] [(<constructor parameters list>)]
[: <block call descriptor> { : <block call descriptor> . . . }]
DELETE_BY <variable destroying descriptor> {, <variable destroying descriptor> }

which differ from **build** and **delete** by the following: body of data will be created or deleted correspondingly by constructor or destructor, but not by commands.

These statements allow hide the body creation process. So, we can use descriptions of data in application, but really create body of data in loadable and closed modules.

For supporting of these methods and more effective usage of distribution of data control procedures on modules we included in Caper the following compiler’s instructions:

1. Extended form of destructors and destructors assignments in structures/collections descriptors

   <constructor> ::= .<identifier>
   <destructor> ::= .<identifier>

   identifiers here are block names.

2. This extended form can be used only with one of instruction of compiler:
   #use module <module descriptor>
   #use_remove module <module descriptor>
   #use block <block name>
   #use struct <structure variable>
   #use collect <collection variable>

   Scopes of these instructions must be finished by

   #nouse

   or new #use

<module descriptor> ::= <type> <module name>
<module name> ::= <file name>

These constructions prescribe to compiler to interpret build, build_by, delete and delete_by by the following schemes:

1. For **#use module** will be execute
   1.1. Loading of selected module.
   1.2. Call of loaded module; result of called module must corresponds to <type> - structure or collection.
   1.3. In the result will be selected member with fixed as Constructor or Destructor name (<identifier> in extended form).
   1.4. Call block, which is referenced in this member with parameters in building/deleting statement.
2. For **#use_remove** all steps of point 1 must be executed and one addition step of removing of loaded module.
3. For **#use block**
   3.1. Call of defined and prototyped block with parameter, which points to variant of calling (from build/build_by or from delete/delete_by instruction.
   3.2. Returned value must be structure or collection (defined by prototype).
   Farther steps are 1.3 and 1.4 of p.1.
4. For **#use struct** 1.3 and 1.4 of p.1 will be executed by <structure variable>.
5. For **#use collect** 1.3 and 1.4 of p.1 will be executed by <collection variable>.

Constructors and destructors can be mixed types: constructor is extended type, but destructor is direct reference to block, and otherwise. As example we show the block usage variant.

Example:

```
struct Methods{ <block> create,  =>
                <block> delete,  =>
                . . . }
internal  <Methods> ModuleLoader ;* block prototype; block returns <Methods> structure

#use block  ModuleLoader
   struct Window { . . . }  {  .create,  .delete } ;* constructor and destructor of this structure
                                                   ;* will be returned by ModuleLoader (Methods)
// This block will be called every time, when Window structure will be built or deleted
   flick ModuleLoader ( build_delete )      ;* Parameter value signs of call from build or delete
     private <Methods> myMethods
     static  build myMethods
     myMethods.create := SomeBlock1   ;* sets SomeBlock1 as Constructor
     myMethods.delete := SomeBlock2   ;* sets SomeBlock2 as Destructor
     return myMethods   ;* returns structure with constructor and destructor references
   endflick
#nouse
```

In this example ModuleLoader returns Methods structure with “create” and “delete” members, which contain references to procedural blocks – Constructor and Destructor and which will be called by building and deleting statements:

```
build private <Window> myWindow( 10, 20, 100, 200 )
delete myWindow
```

which will be compiled by point 3 and, in fact, will be realized

ModuleLoader ().create(10, 20, 100, 200) for call of constructor;
ModuleLoader ().delete() or ModuleLoader ().delete(. . .) for call of destructor.

Construction and Deleting of data can be fully hidden during using build_by and delete_by. It possible create a few interface structures between applying compound data modules and modules. which are servicing these data.

All building instructions have integrated forms with declaration statements.

build private . . .
build_by private . . .
build local . . .
build_by local . . .
build public . . .
build_by public

Example:
build private <tagSomeStru> [10,10] arrOfStru (10, -0.5) : AddConstructor(100), =>
<tagSomeStru> stru(20, 1.75)

build local <tagSomeStru> stru2 (10, -0.5), <tagSomeStru> stru(20, 1.75)

etc.
Arrays in Caper can be multi-dimensional, and according to type can have a compound structure. So, if define

private <var> [10, 20] arrOfVariables, <**string**> [10] arrOfStrings

then every element of array can be any type:

arrOfVariables[1, 1] := 1.25 ;* float is assigned
arrOfVariables[1, 2] := “string” ;* string as array’s element
arrOfVariables[1, 3] := ‘A’ ;* symbor
arrOfVariables[1, 4] := null ;* undefined member of element
arrOfVariables[1, 5] := SomeBlock ;* block’s reference is assigned
arrOfVariables[1, 6] := arrOfStrings ;* reference to array is assigned
etc.
Description and building of arrays in Caper also can be described and built differently

private <int> [] arrOfInt, <SomeStruct> [10, 20, 30] arrOfStruct
. . .
build arrOfStruct
. . .
build arrOfInt(20, 10, 30) : SomeInitFunction(. . .)

or in integrative form:

build private <int> [20, 10, 30] arrOfInt, <SomeStruct> [10, 20, 30] arrOfStruct

In this form both arrays will be described and built immediately.

10. Dual Interpretation of Prototypes

We can consider now what means “dual interpretation”. Caper’s compiler accepts declarations in **internal** and **prototype** statements not only as prototypes of procedural blocks, but and structural data declarations. So, compiler fixes special structural element any declaration of prototype, which contains two mandatory members “body” and “return”, and others members are equal to parameters descriptions.

Example:

```
internal <null> Construct( <int> x, <float> z )
```

will be fixed

```
struct Construct{  <block> body,    =>
                   <null>   return,  =>
                   <int>    x,       =>
                   <float>  z        =>
                 }
```

where "body" contains the reference to block Construct, "return" has "undefined" type (as in prototype). For

```
prototype <great> [10] TypeOfBlock ( <byte> [] bytesArray, <int> x, y )
```

will be fixed

```
struct TypeOfBlock {  <block>          body,       =>
                      <great> [10]     return,     =>
                      <byte>   []      bytesArray, =>
                      <int>            x,          =>
                      <int>            y           =>
                   }
```

but "body" member will not be initialized (obviously, because TypeOfBlock isn't block name, but the name of class of blocks).

Caper allows create variables of generated types by prototypes.

```
private <TypeOfBlock> var, <Construct> var2
```

and use them as following

```
var.bytesArray := someArray  ;* assignment of values for 1-3 parameters
var.x        := 10
var.y        := 20
var.block :=  SomeBlock
var()             ;* call of block SomeBlock with parameters someArray, 10, 20
```

The same result we can have by

```
var.block :=  SomeBlock
var(  someArray, 10, 20 )
```

In fact, both these forms are equal to

```
SomeBlock( someArray, 10, 20 )
```

For var2:

```
build var2
```

and use

```
var2.x := 100
var2.z := 20.0
var2()
```

or

```
var2( 100, 20.0 )
```

In the last case we don't need to initialize of "body" member: this member will be initialized by the **build** instruction. The reference to Construct block will be set as value of this member.

This conception is oriented to the following usage. In parallel computation often we need start a mass of the same procedures (blocks) with different parameters, or different procedures with the same structure of parameters. In fact, Caper supports means for preliminary preparing of procedural calls and start these procedures at the time when it's needed (DO statement; more information about usage is represented in chapter 12).

11. Control Statements: Iterators and Alternatives

Caper has both traditional control constructions and unique ones.

```
IF  <expression>
. . .
[ ELSEIF  <expression> ]
. . .
[ ELSEIF  <expression> ]
. . .
[ ELSE  ]
. . .
ENDIF
```

The SWITCH construction has some differences from ordinary:

```
SWITCH < expression 0 >
[ CASE [ < expression 1 >  | < string or numeric literal > ]  ]
  . . .
[ CASE [ < expression N > | < string or numeric literal > ]  ]
  . . .
ENDS
```

There can be many 'CASE' statements with an empty expression in the right part. For such cases CVM enters the CASE body unconditionally. CASE
with empty expression can be placed in any place.
BREAK statement prescribes to abandon SWITCH.
Exiting from SWITCH / CASE realized by BREAK statement (see below).

Examples:

```
private var1:=0, var2 := NULL, var3 := func(x,y)
 . . .
var2 := GetCase( var3 )
if  var2 == NULL
    var2 := "Hello"
elseif var2 == 1
    var2 := "Good-bye"
elseif var2 == 2
    var2 := "Hi"
endif
```

or the same

```
switch GetCase( var3 )
case NULL
  var2 := “Hello”
  break

case 1
  var2 := “Good-bye”
  break

case 2
  var2 := “Hi”
  //  break statement is not needed.
ends
```

If <u>BREAK</u> statement is absent, then program execution will be continued from current <u>CASE</u> body to the next <u>CASE</u> body.

Caper has traditional iterators:

<u>WHILE</u> <expression>
. . .
<u>ENDW</u>

which iterates body while <expression> returns true value.

<u>REPEAT</u>
. . .
<u>UNTIL</u> <expression>

which iterates body until <expression> will not become true.

All iterations can be terminated by

[IF <expression>] <u>BREAK</u>

or continued by

[IF <expression>] <u>CONTINUE.</u>

Examples:

```
private var1:=0, var2 := NULL, var3 := func(x,y)
 . . .
var3 := 10
while (var1 += 1) < 100
    var3 += func2( ) * func3(var1, var3)
endw

repeat
   var3 := func2() + 100
   if var3 == 10 break
until  var3 < 0
```

12. Calls of Blocks and Parallel Starts

Caper allows start parallel, in fact, by any parallel scheme by Flynn (procedural level): Multiple Procedure Single Data, Single Procedure Multiple Data, Multiple Procedures Multiple Data (MPSD, SPMD, MPMD).

The simplest call of any commands block can be realized by the traditional notation:

<block_name>([<parm1>, <parm2>, . . . , <parmN>])

<block_name> is identifier.

But more general statement of block calls (DO-notation) is

DO [SEQ | SYNCH | ASYNCH] bl1,bl2,...,blK
[WITH quant1,quant2,...,quantL]
[WITHIN med1,med2,...,medK]

where bl1, bl2,..., blK is a blocks descriptors list, and where every bl_i has the following possible forms:

1) <bl_name> [([<parm1>, <parm2>, . . . , <parmN>])] or
2) (<bl_name>, <bl_name2 >, . . . ,<bl_nameP>) ([<parm1>, <parm2>, . . . , <parmN>])

<bl_name> is a name of block or a name is represented by variable. In case 1) *bl* is a block name with possible parameters. Case 2) describes the start of blocks bl_name1, … , bl_nameP with the common pool of parameters. Parameters number in calls of functions and blocks can be varied.

quant1, **. . .**, quantL is a list of quanta (time steps) for every starting block (parallel process).

med1, **. . .**, medK is a list of computers names (identifiers) of multi-computer association or processor identifiers on which the corresponded block will be executed.

The SEQ demands a sequential execution of blocks included in the list. This command will be ended when all blocks from the list are terminated. Quanta are ignored in this case. SEQ can be omitted.

The SYNCH defines a parallel execution of enumerated blocks. Calling procedure will be halted until all the blocks are terminated (synchronous call).

The ASYNCH defines a parallel execution of enumerated blocks, too. But calling procedure execution will be continued (asynchronous call). The last living process will inherit the results of other terminated processes. Then Caper machine sets its own regime as sequential (turning off the parallel submachine). In asynchronous call with time quanta we must set time quanta for all called processes and a quant for calling process.

Example:

```
do synch  (proc1, proc2)( x, y, i, z ) , proc3( x, j ), =>
              proc4(100), proc4( 110 ), proc1( 1, 2, 3, 4 )
```

starts proc1 and proc2 with common parameters area, two different processes proc4 with different parameters and proc1 with separate parameters.

The following form is oriented on usage of dual interpretations of prototyped blocks. We can use arrays of structures for descriptions of started blocks and parameters.

DO [SEQ | SYNCH | ASYNCH] ARRAY <array of prototyped data> {, <array of prototyped data> }

Example:

```
struct Color { <byte> red, <byte> green, <byte> blue )
prototype <word> ColorizeBlock( <Color> color, <int> i )
internal <int>  SomeBlock( <byte> bt, <word> color  ), =>
          <as ColorizeBlock>  SomeBlock2

build private <SomeBlock> [100] arrOfProc, < ColorizeBlock> [100] arrOfProc2
```

```
  private <int> i := 0
  while (i+=1)  <= 100
     arrOfProc[i].bt      := 'A' + i -1              ;* initialization of members
     arrOfProc[i].color := 0xFF0000 + 8*I
     arrOfProc2[i].color.red      := 0xFF'B
     arrOfProc2[i].color.green  := 0xA0'B
     arrOfProc2[i].color.blue    := 0'B
     arrOfProc2[i].i := i
     arrOfProc2[i].block := SomeBlock2         ;* such assignment is needed
  endw
// asynchronous parallel start of 200 processes, which are described in 2 arrays
  do asynch array arrOfProc, arrOfProc2
```

Caper allows conditional call:

IF < expression > <DO-notation>

Every block or function execution will be terminated by ENDBLOCK, ENDFUNC and ENDFLICK statements. In both cases returned value is NULL. For the immediate termination of a blocks we can use

[IF <expression>] RETURN [<expression>] [TO <block name>]

These statements return a value to the calling block or to another block, which occurs earlier in the call stack. If TO <block name> is omitted, then CVM returns a value to the calling block. If we returns from MAIN block or from a single executable block, then CVM returns to OS shell.

For parallel executable processes the internal array of the returned values is created. The returned values are placed into the array in place with index, which corresponds to the parallel process identifier. We can get this value by function RetValue(<proc. id.>), <proc. id.> - here and further in paper is a parallel process numeric identifier.

Unconditional return to OS shell is realized by QUIT statement. In this case all processes, local variables and parameters will be eliminated by *Caper* virtual machine.

13. Labels and Jumps

Caper has two types of label: local and global. Local label can be represented:

<identifier>**:**

Global label must be represented:

<identifier>**::**

Conditional or unconditional local and global jumps are realized by statements:

[IF <expression>] GO <label> for local labels and
[IF <expression>] GGO <label> for global labels

Examples:
In this examples local and global jumps are represented.

```
Block  blk1
  . . .
glLab::  y := 0   ;* global label
  x := 100
```

```
lab1:  y += x +1 ;* local label
  . . .
  go lab1           ;* unconditional jump to local label
  . . .
  if y < 1000 go lab1  ;* conditional jump to local label
  . . .
 endblock

block  blk2 static ( parm1, parm2 )
  . . .
  ggo  glLab     ;* jump to global label
  . . .
  if parm1 > parm2 ggo  glLab     ;* jump to global label
  . . .
endblock
```

14. Caper Virtual Machine

Caper virtual machine (CVM) can start in three different ways:

- start object module, which implanted in Caper executable code.
- load a source code file (source code module);
- load an object module.

If we are using source code module, then *Caper* compiles this code and create in memory the first executable module and block (every module is represented by corresponding block). This module is named automatically as MAIN.

If it's object module, then this module will be loaded and named as MAIN automatically.

Implanted module and block has the MAIN name, too.

In fact, *Caper* machine consists of three sub-machines (a sequential machine, a parallel machine and events machine).

Caper parallel machine executes command-by-command each of started parallel processes (in fact, cooperative scheme of computation). The switching from a parallel process to other CVM carry outs after every command of executed blocks or in special points, which defined by program commands or compiler commands.

All asynchronous events are accepted by *Caper* events machine, which starts an events processing block at the special control moment called as "a virtual machine step". If an event processing block was set, then the machine calls this block in the current parallel branch or initiates a new parallel branch for this block. Every parallel branch has its own call stack. Parallel branches can be halted or broken.

CVM has three regimes: PRIMARY, when CVM dominates over operating system (OS), SECONDARY, when CVM activity depends on OS, and SOLE, when OS is suppressed by CVM to monopolize computer resources.

15. Commands of Compiler.

The set of compiler commands is a small. All commands are executed immediately by compiler, but not by separated preprocessor.

The “feeble” possibilities of preprocessing have own reasoning. In first, Caper has possibilities of dynamic compilation and loading of object modules. Pragmatics and experience of programming in previous versions of Caper show that these two means of the language are sufficient, because the alternatives of source code can be compensated by dynamic selection of needed variant. It’s mean that we can prepare different variants of program codes or object modules. We can choice needed variant during execution of program. Correspondingly, macro-definitions in Caper are utilitarian. They can used as means for designation of numeric and string constants or for fragments of expressions. There are:

#macro <definable> <determinative>

<definable> - the string without space;
<determinative> - the string, which abounded by the end of logical string.

Examples:

#macro Constant_HEX_TEN   0x0A'B
#macro Const_BIN_TWELVE   0b00001100
#macro Literal_Str   "Literal string"
#macro Get_Key   GetKeyb()

```
  . . .
 var :=   Constant_HEX_TEN + Const_BIN_TWELVE
  . . .
 var := Get_Key
```

The following two commands predefine of compilation regime.

#flow [<step>]
  . . .
#endflow

<step> - the number.
These commands define critical range (critical fragment) of statements of program, where all resources of CVM will be monopolized: others parallel threads will be locked until critical fragment will not be ended. <step> defines frequency of passing control to CVM for unlocking events and other parallel processes. It's mean, that after every <step> commands will passing control to CVM. If <step> omitted, then step value is 0 and, correspondingly, the passing of control to CVM will not be executed from critical fragment.
In fact, #flow and #endflow are logical brackets for selection of critical fragment.

Examples:

```
block BLOCK_NAME( fVar1, fVar2)
  . . .
 #flow
  var1 := (var2+var3)/var3
  var2 := (var1+var3)/var3
 #endflow
  . . .
endblock
```

For a forced passing control to CVM we can use

<u>PASS</u> [<process ID>]

where <process ID> is numeric ID of parallel process, to which CVM must pass control.

Examples:
This is a part of program

```
   . . .
#flow
  while (ii+=1) < 100
     x += ii
  endw
#endflow
```

. .

will lock passing control to CVM, in fact events processing and other pseudo-parallel processes will be lock.

The statement

#include <path and file name>

is interpreted traditionally: pointed file will be included in the source code.

16. Events and Virtual Machine

Caper has very powerful means for asynchronous events processing. All events are divided into the following groups: logical events, program events, virtual machine events, operating system events, devices events.

The main construction is a waiting command for asynchronous events

WAIT < event > [BY <block name>]

This statement halts the program or the current parallel process while the expression value is false. The waiting time can be accompanied by block execution: if <event> is false, then <block name> will be started. If <event> is true, then will be executed the next command after WAIT.

Logical events and their processing blocks can be defined by

WHEN <event> <DO-notation>

<event> - any logical or arithmetic expression. This command prescribes to execute the blocks when logical expression value is true. WHEN-events settings are supported by control functions, which activate and deactivate events processing.

WHEN and WAIT combination allows to introduce very comfortable style of programming.

All types of events are supported by collection of functions united by a common style of setting, "freezing", "defreezing" and deleting events.

Caper supports the following style of events processing: we can describe events and set block, which will processing these events. Caper's VM will call events processing blocks every time, when events will be occurred.

The style of setting events processing block is shown on the example of mouse events processing (VM function):

SetMouseEvnRgn(y0, x0, y1, x1, bl_name, filter, procId, ownValue),

where y0, x0, y1, x1 are display region coordinates; bl_name – processing block name; filter parameter is a filter for mouse events; procId – the parallel process identifier, in which processing block must be started by sequential call (CVM switches parallel branch), or this parameter demands to call pointed block as a new parallel process; ownValue is a value of any type, which is set by programmer and registered by events machine; CVM returns this value to the processing block as one of parameters. Similar functions support keyboard, timers and other events.

17. Parallel Computation Control

Caper has a lot of facilities to support parallel processes control and interactions.

In first, *Caper* has abilities to deactivate, activate or broken parallel processes by means of the following statements (every statement has equivalent function):

STOP <proc. id. 1>, <proc. id. 2> , . . . , <proc. id. N>

ACTIVATE <proc. id. 1>, <proc. id. 2> , . . . , <proc. id. N>
BREAK PROCESS <proc. id. 1>, <proc. id. 2> , . . . , <proc. id. N>

OTHERS keyword can be used instead of <proc.id.> to stop, activate or break all processes except the current one (this case for functions is regulated by parameter).

Stopped processes can remain not activated (in fact, all other processes can be terminated; e.g. there will be no active process that could activate stopped ones). In this case the program will be terminated and all stopped processes will be deleted by *Caper* machine (just as local and private variables, input parameters and others).

All parallel processes can be terminated also by

PARABREAK [TO <proc.id.>]

This command has different interpretation for synchronous and asynchronous regimes. In synchronous regime PARABREAK terminates all processes except the calling process and TO <proc.id.> will be ignored. In asynchronous regime all processes excepting <proc.id.> will be terminated.

Caper allows to control the moment of switching over to the next parallel process. At these moments we can call a selected block or function: SetNext([<block name>]) sets the block, which will be called at the switching moments. Such setting can be frozen, de-frozen, changed or deleted.

18. Dynamic Compilation and Loading

Caper has the following constructions for dynamic compilation:

COMPILE [FILE | BLOCK] <file/block name> [IN <block name>]

or

CompileFile(<file name>, <block name> [, <replacement> [, <saving file>]])

which allows to compile a source code from a file or block and to place compiled code into the selected block or create a new block with the compiled code and where

<file name> is a file with a source code to be compiled;
<block name> is the name of a block which is a target for the compiled code;
<replacement> sets a regime for the replacement of existing blocks with new ones; otherwise, if the block name exists then Caper compiler will initiate an error.
<saving file> is the file name in which compiled code will be saved as a module.

The main means for loading and removing object modules are following:

IMPORT <file name> AS [<AS type>] <block name>

loads Caper's module as block with internal block name. <type> is prototype defined name.

REMOVE <block name>[, <block name> ...] | ME

removes blocks from memory. The variant REMOVE ME removes current executed module, in which this statement is placed.

LoadModule(<file name> , <block name> [, <replacement>]) loads the Caper object module from a file as a block with <block name>. <replacement> has the same meaning as in CompileFile. The following statement and function

DELETE BLOCK <bl_name1>, . . ., <bl_nameN>

or

DeleteBlock(< bl_name1>, . . . , < bl_nameN>)

deletes the selected block (which can be a module) with sub-blocks.

CompileCommand(<string>) compiles given string, creates Caper machine code and returns a special pointer-identifier to it. DoCommand(<pointer-identifier>) executes pointed command. DelCommand(<pointer-identifier>) deletes a pointed command.

19. Debugging and Errors Correction

Caper allows debug programs by different ways. Main means are special statements and functions of CVM, which make available information about internal state of VM and current executable program.
#debug and #nodebug are statement of compiler, which allow select program's (module's) parts for compiling in debugging style.

Example:

```
  . . .
#debug
  while (ii+=1) < 100
    x += ii
  endw
#nodebug
  . . .
```

We can write own program module for debugging and assign this module or some block from this module as debugger by VM function

SetDebugger(<block name>, <process ID>, <debugger's folder>,
<debugger's window position and sizes>)

where <block name> is pointing to debugging block. This block must have special description for accepting as parameters all information from VM.

Such realization in Caper allows include a different modules and blocks for debugging for different situations and different parts of program.

<debugger's folder> is folder with files of debugger, <debugger's window position and sizes> is debugger's window coordinates representation.

We can regulate the regime of debugger execution by

SetDebugStatus(<regime ID>)

These regimes include debugger stopping and debugger activation states, and some others. Besides, Caper VM allows take information about calculation process without call debugger procedure and stopping of any other processes ("spying" regime). It allow make parallel process, which will spy for others processes without debugging regime.

Caper VM allows set special block for errors debugging.

SetErrorBL(<block_name> [, <module name> [, <regime>]])

sets block, which will be called every time, when error will be occurred. This block can be part of some module (if <module name> is set; module name in fact is module file name). In this case before block calling module will be loaded, and only after this step block will be called. <regime> is numeric identifier of error processing style.

Caper has a special means for debugging of compilation errors and warnings during compiler work.

20. Interaction Processes by Data and Events

The interaction between parallel processes in Caper can be realized by common for block and his sub-block privet variables (they are visible in block and sub-blocks), places, common parameters (see variant MPSD. Events are their processing means also are communication means.

CVM supports a set of events, which can asynchronously inform about parallel computation states: whether the parallel process was started, stopped, terminated and so on.

References

1. Vartanov S.R. Caper programming language. Preprint 97-5. National Academy of Sci. of Ukraine, Glushkov's Institute Of Cybernetics. Kiev, 1997, 28 pp.
2. Vartanov S.R. The foundations to conception of multi-languages. In: Information Technologies and Management, Vol. 4-3, Yerevan, 2005, 8-23
3. Vartanov S.R. On Parallel Programming Language Caper. Lect. Notes in Computer Sci., HCPN-2001, 501-503.
4. Vartanov S.R. Parallel Programming Methods in Caper Language and its Application in Image Processing. In: SCI2002/ISAS2002, Orlando, USA, Vol.XI, 2002
5. Vartanov S.R. A Mass Parallel Starts In Parallel Programming Language Caper. In: Information Technologies and Management. Vol. 4-1, Yerevan, 2006, 11-18.
6. Vartanov S.R. Parallel Programming in Caper. In: Mathematical Questions of Cybernetics and Computing Technique. Vol.22, Yerevan, 2001, 100-112.
7. Vartanov S.R., Nuridjanyan Sh., R., Hakobjanyan V.A., Manukyan A.G.: Eratosthenes Sieve Parallel Algorithms and Their Programming. In: Information Technologies and Management. Vol.1, 2004, 13-22.
8. Vartanov S.R. The Language and Programming Methods of Image Processing Tasks. Yerevan, 1989, 17 pp.