Mathematical Problems of Computer Science 22, 2001, 100–112.

Parallel Programming in Caper

Sergey R. Vartanov

Chief Research and Project Manager,
Memco Inc. Greenwich, USA.
sergey@memcosoft.com

Abstract

Caper is a parallel programming language, which supports declarative parallel computations and control of all architectures by Flynn [1]. Caper is based on virtual machines system, including own parallel virtual machine. This property allows to create programs not depended on multitasking management of operating systems. 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 to regulate usage of variables by different parallel processes.

1 Introduction

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

The version presented here is the third version (see [2], [3]).

Caper is based on the virtual machine (CVM), which supports both sequential and parallel programs performing. *Caper* commands provide synchronous and asynchronous launch of program procedures with using *Caper*'s own model of pseudo-parallelism, called as "command-by-command" [2] and the familiar model with time quantum. *Caper* allows the start of multiple parallel procedures with common data, a single procedure with multiple data, multiple parallel procedures with multiple data.

Caper is portable. It is independent of what calculation model (sequential or parallel) 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.

From the very beginning the possibility of program self-organization while running was put in *Caper* to adapt to the load and running of the program to calculation needs. *Caper* allows dynamically changing a single command, or fragments of a running program, removing separate object modules, compiling a source code and performing it.

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

Caper provides the possibility to describe various events and to assign the procedures of asynchronous sequential or parallel processing. The language machine distinguishes classes and subclasses of events. The language is provided with multiple facilities of events management: to define events, freeze, defreeze, remove and initialize them.

All *Caper*'s variables are polymorph and differ in scope, the way created and its time basis. Besides usual variables *Caper* also has resources to manage variables. They are characterized by states or instances, which can then be changed at any given moments of calculation. This allows the regulation of access to the variables, and initialize calculation when the variables acquire certain states.

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.

2 Program Structure

The following *Caper* constructions will be represented by short descriptions.

Caper is founded on ASCII characters set. We differentiate logical and physical strings. A physical string is the string of a text editor. A logical string is a string occupying 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 few logical strings. *Caper* has signs for comments definition: '*' and '//' for a logical comment string, '/*' and '*/' - comment brackets, and some other variants.

Caper has two types of label: local and global.

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.

A block is a general aggregate of executable commands and different data.

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

<blockname> is an identifier.

STATIC keyword prescribes to compiler to place the parameters into the module body.

<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. If a block has IMAGE type

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

the block body is then loaded from the file <filename> in a compilation step.

Every block is 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 of two types: machine (or environment) functions and programmed functions, which are defined by programmer. The machine functions are intended for CVM call. The blocks of commands of the two special types are functions in *Caper*:

```
FUNCTION <block name> [(<fparm 1>, <fparm 2>, ..., <fparm N>)]
...
ENDFUNC
```

and

```
FLICK <block name>
...
ENDFLICK
```

Source code compilation result is *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.

All functions are compiled as critical fragments.

Functions can be started by DO-statement or by a traditional notation. Functions are differed from usual blocks in the following properties:

- all functions are compiled only as uninterrupted blocks (see #flow compiler statement);
- "flow" and "endflow" statements don't change the regime of compiling.
- functions return values directly and can be assigned to variables.

FLICK is a type of function which has all function properties and additionally changes CVM regime on SOLE – blocking events in the sub-machine of CVM. Any return from flick changes CVM regime and restores the CVM regime.

Since block names can be public and visible in the entire program, to differentiate the name of the block in various modules the word INTERNAL is introduced to limit the scope to the given module and is represented as follows:

```
INTERNAL <block name1>, <block name2>, ..., <block name N>
```

3 Variables and Places

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

Caper doesn't permit direct access to computer memory.

The strings of *Caper* are arrays of bytes (symbols), which can be addressed as arrays elements. At the same time, string can be considered as a whole unit of data.

Places of *Caper* have a special meaning. Every place-variable has one of the following statuses: WRITE_ONLY, READ_ONLY, LOCKED, FREE or UNLOCKED (synonyms). *Places* usage is controlled by CVM.

At first all defined *places* have FREE status. A *place* owner is the block, which is the first to set a not FREE *place* status. Any attempts to use a *place* or its value from the block that is not the *place* owner and conflicts with the *place* status will cause an internal CVM error.

The control of place is a mean of controlling the resource, which is represented by this place. All *Caper* has various variables with multitude of scopes; the period in which the variable is created and the length of its operational life. There are Public (global), Private and Local variables. The statements of variables definition are formed by

{ PUBLIC | PLACES | PRIVATE | DEFINE | LOCAL } <var. list with initialization>

<var. list with initialization> is assignment expressions list.

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

Private variables that were defined in a block are visible only in the block and its sub-blocks. If private variables are defined outside of module blocks then their scope effects the entire program module. Private variables are static. They are placed in object module's body. These variables are being deleted with the module's body. DEFINE statement forms dynamic private variables at the moment of module's execution. This definition ensures the ability to reenter in the module's blocks, such as in the case where every parallel thread creates its own copy of private variables pool.

Local variables scope is only a block body. 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 public-, private-, define-variables are lead by <u>INITIAL</u> keyword then repetitive initialization of variables is prohibited.

LockF(<place name>, <status>) assigns the status of a *place*. <status> is one of above enumerated statuses. *Caper* allows defining 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.

4 Control Statements: Iterations and Alternatives

Caper has both traditional control constructions and unique ones.

<u>IF</u>

```
...
[ ELSEIF ... ]
...
[ 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. BREAK statement prescribes to abandon SWITCH.

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

```
[ IF <expression> ] GO <label>  and  [ IF <expression> ] GGO <label>
```

Caper has traditional iteration means:

```
WHILE
...
ENDW

REPEAT
...
UNTIL.
```

All iterations can be terminated by

```
[ IF <expression> ] BREAK
```

or continued by

```
[ IF <expression> ] CONTINUE.
```

```
FOR <initialization 1>, <initialization 2>, . . ., <initialization K
[ TO <condition 1>, <condition 2>, . . ., <condition L> ]
[ BY <step 1>, <step 2>, . . ., <step M> ]
DO <DO-descriptor>
```

introduced in Caper for optimized calls of the parallel processes and will be described further.

5 Calls of Blocks and Parallel Computations Initiation

The simplest call of any commands block (including FUNCTION and FLICK) is realized by the traditional notation:

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

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 ]
```

and a conditional call

```
IF < expression > DO <DO-descriptors>
```

where bl1, bl2,..., blK is a blocks descriptors list, which 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 contained in a variable. In case 1) *bl* is a block name with possible parameters. In case 2) the start of blocks bl_name1, ... , bl_nameP with the common pool of parameters is described. Parameters number of functions and blocks call can vary.

quant1, . . ., quantL is a list of quanta for every starting block.

med1, . . ., medK is a list of computers names (identifiers) of multi-computer association or processor identifiers the block will be executed in (it was described in detail in Caper-2 [2] and it's not a theme of this paper).

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. Quantum 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 the calling procedure for 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 quantum we must set quantum for all called processes and a quantum for calling process.

Every block or function execution will be terminated by ENDBLOCK, ENDFUNC and ENDFLICK statements. In all cases the returned value is NULL. The immediate termination of a block or function is accomplished by

```
[ 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 return 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 on 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, parameters, defined private variables will be eliminated by *Caper* machine.

```
FOR <initialization 1>, <initialization 2>, . . ., <initialization K
DO <DO-descriptor>
[ TO <condition 1>, <condition 2>, . . ., <condition L> ]
[ BY <step 1>, <step 2>, . . ., <step M> ]
```

Supports the parameters preparation beginning with initial values to stated conditions and change in the value by <step> expressions .
<initialization 1>, <initialization 2>, . . ., <initialization K> are assignment expressions.
<condition 1>, <condition 2>, . . ., <condition L> are logical expressions,
which abounds FOR statement execution: calculation will be stopped, if a minimum of one of the conditions is false.
<step 1>, <step 2>, . . ., <step M> are expressions which change variables values.
<DO-notation> represents the starting blocks.

FOR statement has the following interpretation: after initialization <step> expressions will be executed until <condition> expressions are true. During the execution CVM forms parameter pools for all starting blocks.

Example of parallel starts in Caper:

```
Private i, j, k

for i:=1, j:=10, k:=100                                                    =>
do asynch proc1( x, y, i,  z ),  proc2( z ),  proc3( x, j ),  proc4( k )  =>
to  i <= 10,  j<100,  k < 1000                                             =>
by i += 1, j += 1,  k += 10
```

will form the list of starting processes

```
proc1( x, y, 1,  z ), proc2( z ), proc3( x, 10 ), proc4( 100 ),
proc1( x, y, 2,  z ), proc2( z ), proc3( x, 11 ), proc4( 110 ),
proc1( x, y, 3,  z ), proc2( z ), proc3( x, 12 ), proc4( 120 ),
. . .
proc1( x, y, 10,  z ), proc2( z ), proc3( x, 19 ), proc4( 190 )
```

and start them when a minimum of one of the conditions is false (40 parallel processes in this case).

The one of the ways of processes multiplication is:

```
j:=10, k:=100
i := 0
while ( i+=1 ) <= 10 && j<100 && k < 1000
    do asynch proc1(x, y, i, z ), proc2( z ), proc3( x, j ), proc4( k )
    j+=1, k+=10
endw
```

The other example of parallel start:

```
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 yet separate parameters.

6 Caper Virtual Machine

Caper virtual machine can start in three different ways:
- starting object code implanted in Caper executable code.
- loading a source code file (source code module);
- loading an object module.

If it is source code module, then *Caper* compiles this code and creates the first executable module and block (every module is represented by corresponding block). This module is named automatically as MAIN.

If it is object module, then the module will be loaded and named as MAIN automatically. Implanted module and block have the MAIN name, as well.

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 started parallel processes. The switching from a parallel process to other CVM carry out after every command of executed blocks or in special points, which is 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 "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.

7 Events and Events 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.

Logical events and their processing blocks are defined by

WHEN <event> DO <DO-descriptor>

<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.

The second principal construction is a command waiting for asynchronous event

WAIT < event > [BY <block name>]

This command halts the program or the current parallel process while the expression value is false. The waiting time can be accompanied by block execution. 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", "de-freezing" and deleting events. The style of setting events processing block is shown on the example of mouse events processing: SetMouseEvnRgn(y0, x0, y1, x1, bl_name, filter, procId, selfId), 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; selfId is a numeric value, 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.

8 Parallel Computation Control

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

First, *Caper* has the ability to deactivate, activate or break 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.

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 controls 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.

9 Arrays, Strings, Regions, File-arrays and Collections

Caper supports dynamic and static multidimensional arrays, strings, so called "regions" – virtual arrays, which are defined as sub-arrays of created arrays. File-arrays support a work with files as it does with arrays.

Arrays can be of any type, including arrays of collections, strings, arrays, blocks.

Collection is a variables group, which must be described and created dynamically or statically in a module body. Collections descriptions can be redefined during calculations time, collections can be removed from memory.

```
COLLECTION <collection tag> { <variable name 1>,
                              <variable name 2>,
                              . . . ,
                              <variable name N>
                            }
```

<collection tag> - identifier, which represents collection;
<variable name> - collection internal variable name.

Any variable of collection can contain data of any type.
We can create collection by

```
STATIC <collection tag>[ { [ <initialization value 1>,
                             <initialization value 2>,
                             . . . ,
                             <initialization value N>
                           ]
                         } ] <variable 1>, ..., <variable N>
```

or

```
<variable> := <collection tag>{ [ <initialization value 1>,
                                  <initialization value 2>,
                                  . . . ,
                                  <initialization value N>
                                ] }
```

The STATIC keyword prescribes that collection must be created in current module body. In other cases, collection will be created in dynamic memory.
<variable 1>, ..., <variable N> must be defined in any way as Local, Private or Public variables.
<initialization value> initiates the collection internal variable. If <initial values> are omitted, then all variables have NULL value.

To address a variable within the collection is done as follows:

<collection name>.<variable name>

Example of Caper's code:

```
#macro  false   0
#macro  true    1
internal  sumFunc, tstFunc
private arr := array( 'I', 0,10, 20 ), str := string( 200, 'A' )

collection  tagColl{ time, day, month, year, name, function }

block tBlock1(parm1, parm2)
private coll
local  year := 1999

  coll              := tagColl{"10:00", 12, 12, year+1 }
  coll."name"       := "Armine"
  coll."function"   := sumFunc
  year := coll."function"( arr, year )
  coll."day" += 1
  return year  to  Main_block

  func  sumFunc static ( parm1, parm2 )
       private i, j, k:=0
       if   ! tstFunc()  return false
       i := 0, j:= 10
       while ( i += 1) <= j
           k += 1
           parm1[  i, k ]  := i*parm2
       endw
       return k
     endfunc

  func  tstFunc
       if  coll == null  return  false
       return true
  endfunc

endblock
```

10 Processes Interaction by Data and Events

Besides using public resources (variables, *places*, blocks) interaction of parallel and sequential processes by means of local variables and parameters setting and receiving is permitted. All means are grouped by functions of reading and writing local variables and parameters.

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.

11 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 compiling 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.

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 its 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.

IMPORT <file name> AS <block name>

loads Caper's module as block with internal block 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.

12 Resume

Of course, I can't give a full idea of Caper in a short paper, and represent all properties, abilities of the language and fulfill our experience of its usage. I'm giving some examples and characteristics of Caper usage below. So, we have an experience with about 100 thousand CVM's parallel processes in a single processor (Pentiums). There are very natural and effective realizations of such components as pop-up menu with animations (10-20 items), parallel reading and processing files, parallel searching in memory and in databases, etc. (in [3] a few algorithms of image processing were described; their real parallel execution for 7 computers in network was made in Caper-2 and represented in [2]).

Caper continues to develop and expand and realize its full potential.

References

[1] M. Flynn, Ultraspeed Computing Systems. IEEE Trans.Comp., vol. 54, no. 12., 311-320, 1966.

[2] S. Vartanov, On Parallel Programming Language Caper. Lect. Notes in Computer Sci., HPCN-2001, 565-569.

[3] S. Vartanov, The CAPER Programming Language. Preprint 97-5. National Academy of Sciences of Ukraine, Glushkov Institute of Cybernetics. Kiev (1997)

Զուգահեռ ծրագրավորում CAPER–ում

Ս. Ռ. Վարդանով

Ամփոփում

Հոդվածում նկարագրվում է զուգահեռ ծրագրավորման լեզուն – CAPER-ը, որը ապահովվում է բոլոր համակարգային դասերի՝ ըստ Flinni:

CAPER-ը ունի վիրտուալ մեքենաների համակարգ նաև իր սեփական զուգահեռ վիրտուալ մեքենան:

CAPER-ը ունի ինքնակազմակերպման և ասինխրոն պատահարների ծրագրավորման միջոցներ: