

# MC# : a language for concurrent distributed programming based on .NET

Vadim Guzev  
People's Friendship  
University of Russia,  
Russia, Moscow,  
[guzevv@quantumart.ru](mailto:guzevv@quantumart.ru)

Yury Serdyuk,  
Program Systems Institute  
of the Russian Academy of Sciences,  
Russia, Pereslavl – Zalessky,  
[yury@serdyuk.botik.ru](mailto:yury@serdyuk.botik.ru)

## ABSTRACT

In this work we introduce the main ideas lying in the foundation of MC# programming language.

The premises and goals for the language development are presented and new constructs such as movable methods, channels and handlers are described. We give an example of concurrent distributed program on MC# for computing the primes by “Eratosthenes sieve” method.

In the second part of paper we provide a formal basis for the proposed language. The start point is objective join-calculus, which we extend to the distributed object calculus. For the latter, we define an operational semantics in the “abstract chemical machine” style. Then an example of term reduction in this calculus is presented. We conclude with brief description of implementation and directions for the further work.

## Keywords

Concurrent distributed programming, MC#, movable methods, channels, handlers, distributed object calculus.

## 1. INTRODUCTION

At now the wide spread using of computer systems with massive parallelism such as clusters and GRID-architectures posed again a problem of developing high-level, powerful and convenient programming languages which allow one to create complex, but at the same time, robust software systems that effectively use the possibilities of concurrent distributed computations.

The program interfaces and libraries, which are available now, such as MPI ( Message Passing Interface ), in all their varieties, are realized for C and Fortran languages, and so are very low-level and not suitable for modern object-oriented programming languages such as C++, C# and Java.

From the other side, the software tools for Grid-computing such as, for example, Globus Toolkit, are intended to distribute a set of tasks ( applications ) through computational network - not for parallelizing

of single programs as such. Furthermore, it is very difficult to deploy and use them.

In general, modern high-level programming languages consist of two parts :

- 1) proper basic constructs and

- 2) collection of libraries accessible through suitable API ( Application programming Interface ).

New requirements to programmers productivity ( by increasing of the abstraction level of language constructs ) and robustness and security of programs which they develop, caused a migration tendency of most important APIs to corresponding native constructs of programming languages.

So, for example, the introduction of asynchronous methods and chords to Polyphonic C# language [BCF04], an extension of C#, allows one to avoid System.Threading library needed to implement multithreaded applications. From the other hand, the introduction of new data types ( streams, anonymous structures, discriminated unions and others ) along with appropriate tools for query defining to C $\omega$  language [BMS], makes unnecessary ADO.NET data subsystem ( namely, traditional System.Data and System.XML libraries for relational and semistructured data processing ).

We propose to make the next step in the direction mentioned above – to introduce the high-level constructs for developing of concurrent, distributed

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*.NET Technologies'2005 conference proceedings,*  
*ISBN 80-903100-4-4*  
Copyright UNION Agency – Science Press, Plzen, Czech Republic

programs in object-oriented language, and so to avoid a necessity to use by a programmer System.Remoting library needed to create distributed applications.

From the practical point of view, the goal of development of MC# language was a designing a language for industrial concurrent distributed programming allowing to create complex software applications having satisfactory efficiency. This language aims to replace C and Fortran languages in these areas. The choice of C# as a basic language gives possibility to use modern object-oriented programming language equipped with the reach libraries ( for work with the Web, in particularly, with Web-services, for creation graphical applications, for processing XML documents, for implementation of systems with a high degree security and so on ), and, at the same time, to exclude low-level and dangerous features such as the C pointers, which dramatically decrease a programmer productivity and reliability of software systems. Comparing MPI interface and MC# language, in the latter there is no necessity to distribute processes over cluster nodes or Grid-machines explicitly (though such possibility also exists in it) – it is enough to point out which functions (methods) can be executed concurrently. Similarly, there is no necessity to code an object (data) serialization to move them to remote node/machine in MC# programs by explicit way – Runtime-system performs an object serialization/deserialization automatically.

In fact, MC# language is an adaptation of basic idea of the Polyphonic C# language (more precisely, basic idea of the join-calculus [FG02] ) for concurrent distributed computations, because an execution of programs in Polyphonic C# as before is considered by its authors either on a single computer or on many machines interacting through the remote methods calls using .NET System.Remoting library. In MC# case, an execution of autonomous asynchronous method can be scheduled in a different machine selected by one of the two ways: either through explicit indication by programmer or automatically (typically in this case, cluster node or machine in Grid-network is selected due to least workload principle). Interaction of asynchronous methods which are executed on the different machines is realized through channels and channel message handlers, which are defined in MC# program by the chords in Polyphonic C# style.

As our theoretical contribution, we propose an object calculus for concurrent and distributed computations and define a precise operational semantics for it. This semantics is a formalization of the execution algorithm of the MC# Runtime-system.

The paper is organized as follows. Section 2 describes novel constructs of MC# – movable methods, channels and channel message handlers. In Section 3 we demonstrate using of MC# language constructs to develop concurrent, distributed program for computing the primes by “Eratosthenes sieve” method. Section 4 presents the formal semantics of MC# language abstracted to concurrent and distributed object calculi. In Section 5 we give details about MC# language implementation parts of which are compiler and Runtime-system. We provide conclusions and directions for the further work in Section 6.

## 2. NOVEL CONSTRUCTS OF MC# : MOVABLE METHODS, CHANNELS AND HANDLERS

In any object-oriented language, conventional methods are synchronous : the caller always waits until the method called is completed, and only then continues its work.

The key feature of the Polyphonic C# (which, in fact, became a proper part of C $\omega$  language, and from now on we will refer only to the latter – <http://research.microsoft.com/omega> ) is the use of so called “asynchronous” methods in addition to conventional synchronous methods. Indeed, such asynchronous methods are designated to play two major roles in program :

- 1) as autonomous methods executed in own threads, and
- 2) as methods used for delivering data to convenient, synchronous methods.

In MC# language, these two kinds of methods form two special syntactic categories correspondingly:

- 1) **movable methods**, and
- 2) **channels**.

One more special syntactic class consist of channel message handlers (briefly, channel handlers, or even, simply, handlers), which are similar to synchronous methods jointly used with asynchronous methods in chords.

### 2.1 Movable methods

A basic technique of parallel program writing in MC# language is reduced to label by the special **movable** keyword the methods which may be transferred for execution to other machines (which either can be pointed out by a programmer or can be selected by Runtime-system automatically, typically, in order to provide balanced workload over accessible nodes/machines):

```

modifiers movable method_name (arguments) {
    < method body >
}

```

The specifics of movable methods is that

- their call completes essentially immediately,
- they never return a result.

Correspondingly, by the rules of correct definition, movable methods

- may not have a **static** modifier, and
- they never return a result.

To be worth to note, that the objects created during the course of program execution are, by their nature, **static** : once created , they are remained to be bounded to that machine where they “born” (in particular, they are registered by Runtime-system on that machine that is necessary to deliver the channel messages intended for that objects).

The movable method call has two syntactical forms :

- 1) object\_name.method\_name ( arguments );

- in this case, Runtime-system selects a node/machine for movable method execution automatically, and

- 2) machine\_name@object\_name.method\_name ( arguments )

- an explicit indication of machine for execution of given method.

**The first key feature of MC# language** is the following : under movable method call, all necessary data for this method, namely

- 1) the object itself to whom given movable method belongs, and
- 2) arguments for the latter

is only **copied** (but not moved) to remote machine, and as a consequence, all changes which are performed over the copy afterwards have no influence on the original object.

In particular, if an object, which is copied, have channels or handlers (see Section 2.2), they are created all over again on remote machine.

There are two modes for movable methods running : “functional” and “nonfunctional” (objective), which affect on efficiency of MC# program execution. These modes are given by the modifiers **functional** and **nonfunctional** in movable method declaration (the former is a default value).

In functional mode, an object for which movable method called, isn’t transferred to remote machine (i.e., all data needed for movable method are passed through its arguments). By explicit application of

**nonfunctional** modifier, we point out that the object must be passed to remote machine also.

Using MC# on cluster architectures (which typically consist of frontend machine and subordinated nodes) with explicit indication of execution place for movable method call, we must also provide a name of frontend machine along with node name in the form :

```
machine_name.node_name@o.m( args );
```

## 2.2 Channels and handlers

The channels and channel message handlers are tools to support for an interaction of distributed objects.

Syntactically, the channels and handlers are defined by the chords in Cø style. In the following example, channel *sendInt* for transferring single integers is defined along with the corresponding handler:

```

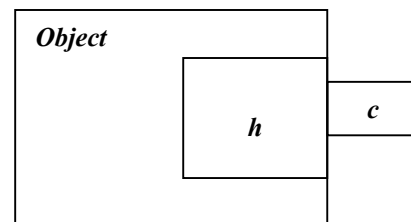
CHandler getInt int() & Channel sendInt ( int x ){
    return ( x );
}

```

In such declarations, handlers are defined with the general format

```
modifiers CHandler handler_name return_type(args)
```

By the rules of correct definitions, channels and handlers may not have a **static** modifier, and so, similarly to familiar methods, they always are bound to some object :



**Fig.1** Object with channel and handler

Thus, we may send an integer *x* on the channel *sendInt* by

```
a.sendInt ( x );
```

where *a* is an object for which the channel *sendInt* has defined.

A handler is used to receive values from the channel (or group of channels) which have defined jointly with it. For example, to receive value from the channel *sendInt* we need to write

```
int y = a.getInt();
```

If one calls a handler and there are no calls to corresponding channel, then the handler call is blocked. When a value is received by the channel, a

body of chord runs (this body may consist of arbitrary computations), returning resulting value to the handler.

Conversely, on a call to the channel, if there are no pending calls to handler, then the call to channel is simply saved to queue, where are accumulated all values from multiple sends on this channel.

As in C $\omega$ , it is possible also to declare a few channels in the single chord with the aim to synchronize them:

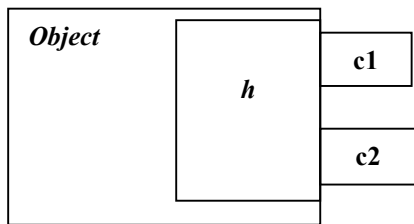
```

CHandler equals bool() & Channel c1 ( int x )
                        & Channel c2 ( int y ) {
    if ( x == y )
        return ( true );
    else
        return ( false );
}

```

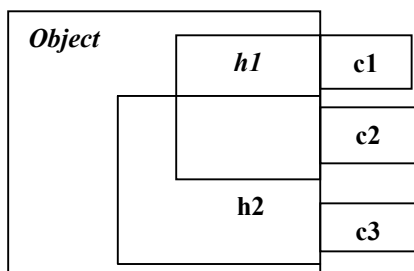
Thus, a general rule for chord triggering is following: body of chord is only executed once **all** the methods (Handler and channels) from chord header have been called (see a formal semantics in Section 4).

Above example represents a case of single handler with few channels:



**Fig. 3.** Object with single handler over two channels

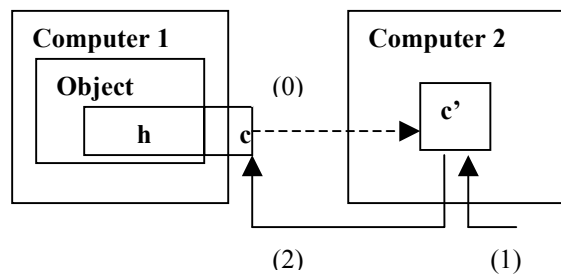
Also it is possible to declare the same channel in different chords:



**Fig.3.** Object with "shared" channel

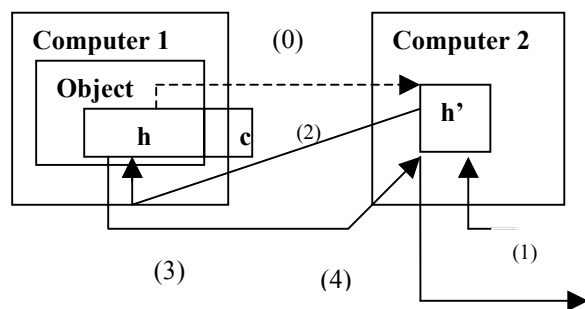
So, once we have values both in channels  $c1$  and  $c2$ , then handler  $h1$  can be triggered. Similar is the case for channels  $c2$  and  $c3$  and handler  $h2$ . All together this, in general, leads to nondeterminism in program behaviour.

**The second key feature of MC# language** is following: channels and handlers may be passed as arguments to methods (in particular, to movable methods) **separately** from the object to which they belong (in this sense, they are similar to **delegate** constructs of C# language). A passing of channels and handlers as arguments of movable method is different from passing of the basic objects: a special structure is copied to remote machine, which play a role of mediator with original channel (or handler) – not the channel (or handler) itself. This replacement has been hidden for programmer – he may use passed channels and handlers on remote machine as if they would be placed locally. Really, all actions over channels and handlers (more precisely, with their "proxies") are transferred to original channels and handlers by Runtime-System (see also formal semantics definition in Section 4).



**Fig.4.** Message sending by remote channel:

- 0) "proxy" creation (passing channel as movable method argument),
- 1) message sending,
- 2) message redirection.



**Fig. 5.** Message reading from remote handler:

- 0)"proxy" creation (passing handler as movable method argument),
- 1) query for message reading,
- 2) query redirection,
- 3) message redirection,
- 4) message return.

### 3. PROGRAMMING IN MC#

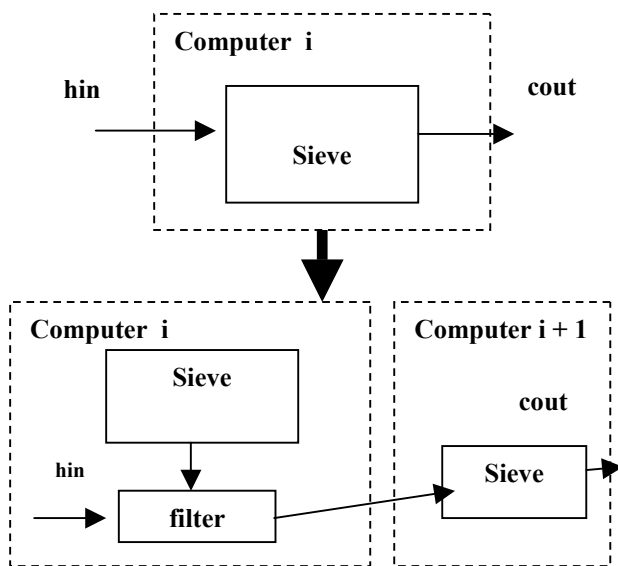
In given Section, we build parallel distributed program for finding primes by the sieving method (another name of this method is “Eratosthenes sieve”).

Given a natural number  $N$ , we need to enumerate all primes from 2 to  $N$ . The sieving method is the following recursive procedure:

- 1) select a first integer from the original list  $[2, \dots, N]$ , namely 2, and output it as first element of output list;
- 2) construct a new list by deleting from old list all integers which are multiplies to the first element of the latter list;
- 3) apply given procedure recursively to the new list.

Main computational procedure, which we call *Sieve*, recursive calls of which will be distributed over computational network, have two arguments: handler *hin* for receiving integers from original list and channel *cout* for producing primes extracted from input list. The end marker in both lists is -1.

Elementary step of unfolding computations (in other words, generation of next unit of the “conveyor”, which sieves an integer stream) is given schematically in Fig. 6.



**Fig. 6.** Unfolding computations in distributed sieve method

We give a full program text in MC# language, where original list of natural numbers is sent on channel *Nats* and resulting list of primes is received from channel *Primes* by handler *getPrime* :

```

class Eratosthenes {
public static void Main (string[] args ) {
    int N = System.Convert.ToInt32 ( args [ 0 ] );
    Eratosthenes E = new Eratosthenes();
    new CSieve().Sieve ( E.getNat, E.Primes );
    for ( int n = 2; n <= N; n++ )
        E.Nats ( -1 );
    E.Nats ( -1 );
    while ( ( int p = E.getPrime() ) != -1 )
        Console.WriteLine ( p );
    }
    CHandler getNat int() & Channel Nats (int n) {
        return ( n );
    }
    CHandler getPrime int() & Channel Primes (int p) {
        return ( p );
    }
}
class CSieve {
movable Sieve(CHandler int()hin,Channel(int) cout){
    int head = hin();
    cout ( head );
    if ( head != -1 ) {
        new CSieve().Sieve ( hinter, cout );
        filter ( head, hin, cinter );
    }
}
    CHandler hinter int() & Channel cinter (int x) {
        return ( x );
    }
    void filter (int p, CHandler int() hin, Channel (int)
        cfiltered){
        while ( ( int n = hin() ) != -1 )
            if ( n % p != 0 ) cfiltered ( n );
        cfilterd ( -1 );
    }
}
}

```

## 4. FORMAL SEMANTICS OF MC#

For more compact and understandable presentation of operational semantics, we

- at first, define semantics only for basic constructs of language, such as object declaration, joints (abstractions of chords) and concurrent composition operator,

- at second, define semantics by two steps: in the beginning, we define it for concurrent (local) object calculus, and then for distribute calculus (where a place of execution – a site – is introduced).

### 4.1 Concurrent object calculus

Let  $N$  be a set of names, which, from informal point of view, will be treated as variable names  $x, y, \dots$ , method names (in particular, channel and handler names)  $m, n, c, h, \dots$ , and object names  $a, b, \dots$ . The symbol  $\alpha$  will be used for name of any type. The tuples of names are denoted as  $\mathbf{x}, \mathbf{y}$  and so on. The letters  $\sigma$  and  $\theta$  (possibly, with subscripts) we use to denote mappings from finite states of  $N$  to ones. If  $\sigma: N_1 \rightarrow N_2$ , then domain ( $\sigma$ ) =  $N_1$  and range ( $\sigma$ ) =  $N_2$ .

The syntax of concurrent object calculus is given below (this calculus is a slightly simplified version of the **objective** join-calculus [FLMR03]):

$$\begin{aligned}
 J &::= m(\mathbf{x}) \\
 &\quad | J_1 \ \& \ J_2 \quad \text{joint} \\
 D &::= J \ \triangleright \ P \quad \text{process with joint} \\
 &\quad | D_1, D_2 \quad \text{composite definition} \\
 P &::= 0 \quad \text{empty process} \\
 &\quad | a.J \quad \text{message sending} \\
 &\quad | P_1 \parallel P_2 \quad \text{concurrent composition} \\
 &\quad | \underline{\text{obj}} \ a = D \ \underline{\text{in}} \ P \quad \text{process with local} \\
 &\hspace{15em} \text{object definition}
 \end{aligned}$$

The concurrent object calculus consist of three syntactical categories: joints  $J$ , definitions  $D$  and processes  $P$ . In this calculus, we generalize a message sending operator

$$a . m(\mathbf{x} \ \sigma)$$

(where  $\sigma$  is mapping from names to names replacing formal parameters by actual values) to operator for sending few messages simultaneously

$$a . J$$

(again for more compact representation of operational semantics).

Furthermore, we explicitly define a concurrent composition operator in the calculus, though this operator, in fact, is defined in MC# only implicitly: each message sending or method call results to

creation of concurrent thread (here we make further simplification that all methods are concurrent).

A definition of single object for each process

$$\underline{\text{obj}} \ a = D \ \underline{\text{in}} \ P$$

isn't really an essential constraint; more general construct

$$\underline{\text{obj}} \ a_1 = D_1, a_2 = D_2, \dots, a_n = D_n \ \underline{\text{in}} \ P$$

we can express by nested construction

$$\underline{\text{obj}} \ a_1 = D_1 \ \underline{\text{in}}$$

$$\underline{\text{obj}} \ a_2 = D_2 \ \underline{\text{in}}$$

$\dots$

$$\underline{\text{obj}} \ a_n = D_n \ \underline{\text{in}} \ P$$

There are two variants of operational semantics definition for concurrent calculi :

1) abstract operational semantics using structural congruence relation, and

2) “chemical abstract machine” style semantics which is more close to actual implementation.

We give both variants for concurrent object calculus along with their simple relationships.

#### 4.1.1 Abstract operational semantics

Let  $\sigma$  be a mapping from  $N$  to  $N$ . The operational semantics of concurrent object calculus is defined, in fact, by a single rule :

$$\begin{aligned}
 \text{REDUCTION} : \underline{\text{obj}} \ a = J \ \triangleright \ P \ \underline{\text{in}} \ J\sigma \parallel Q &\rightarrow \\
 \underline{\text{obj}} \ a = J \ \triangleright \ P \ \underline{\text{in}} \ P\sigma \parallel Q &
 \end{aligned}$$

Because we may have a few joints for given object, we need to define a similar rule in compositional context:

$$\begin{aligned}
 \text{COMPOSITION} : \underline{\text{obj}} \ a = J \ \triangleright \ P, D \ \underline{\text{in}} \ J\sigma \parallel Q &\rightarrow \\
 \underline{\text{obj}} \ a = J \ \triangleright \ P, D \ \underline{\text{in}} \ P\sigma \parallel Q &
 \end{aligned}$$

In general case, to apply these rules we need to rearrange a term structure. For this purpose, as usually, a structural congruence relation is defined, which is a smallest relation  $\equiv$  over processes and definitions, satisfying to axioms ( $\alpha$ -convertibility is treated as in  $\lambda$ -calculus):

1.  $P \equiv P'$ , if  $P$  and  $P'$  are  $\alpha$ -convertible
2.  $D \equiv D'$ , if  $D$  and  $D'$  are  $\alpha$ -convertible
3.  $P \parallel Q \equiv P' \parallel Q'$ , if  $P \equiv P'$  and  $Q \equiv Q'$
4.  $\underline{\text{obj}} \ a = D \ \underline{\text{in}} \ P \equiv \underline{\text{obj}} \ a = D' \ \underline{\text{in}} \ P'$ ,  
if  $D \equiv D'$  and  $P \equiv P'$
5.  $P_1 \parallel P_2 \equiv P_2 \parallel P_1$
6.  $D_1, D_2 \equiv D_2, D_1$
7.  $P \parallel 0 \equiv P$

8.  $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$   
 9.  $D_1, (D_2, D_3) \equiv (D_1, D_2), D_3$   
 10. ( $\text{obj } a = D \text{ in } P$ )  $\parallel Q \equiv \text{obj } a = D \text{ in } (P \parallel Q)$ ,  
 if  $(\{a\} \cup \text{fn}(D)) \cap \text{fn}(Q) = \emptyset$ .

The set  $\text{fn}$  of free names is defined recursively over term structure:

$$\begin{aligned} \text{fn}(0) &= \emptyset \\ \text{fn}(a.J) &= \{a\} \cup \text{fn}(J) \\ \text{fn}(P_1 \parallel P_2) &= \text{fn}(P_1) \parallel \text{fn}(P_2) \\ \text{fn}(\text{obj } a = D \text{ in } P) &= (\text{fn}(D) \cup \text{fn}(P)) \setminus \{a\} \\ \text{fn}(J \blacktriangleright P) &= \text{fn}(J) \cup \text{fn}(P) \\ \text{fn}(D_1, D_2) &= \text{fn}(D_1) \cup \text{fn}(D_2) \\ \text{fn}(m(\mathbf{x})) &= \{\mathbf{x}\} \\ \text{fn}(J_1 \& J_2) &= \text{fn}(J_1) \cup \text{fn}(J_2) \end{aligned}$$

From the above definition, we see that  $a$  in  $\text{obj } a = D \text{ in } P$  and  $\mathbf{x}$  in  $m(\mathbf{x})$  are **bound** names. As usually, we can free to rename bound names for  $\alpha$ -convertibility of terms.

#### 4.1.2 Operational semantics in “chemical abstract machine” style

An operational semantics defined as “chemical abstract machine” is more closer to actual implementation and is given as set of rewriting rules over objects (more precisely, over definitions of named objects) and processes configurations. A configuration

$$\mathcal{D} \mid\!-\! \mathcal{P}$$

consist of a set  $\mathcal{D}$  of named definitions (representing the objects) and a multiset  $\mathcal{P}$  of concurrently running processes. “Chemical reactions” (term reductions) are given by the rewriting rules of two kinds: structural rules  $\equiv$ , making term rearrangement and reduction rules  $\rightarrow$  representing computation steps.

Below, a notation  $a = [J \blacktriangleright P]$  is an abbreviation of object declaration which includes a process with a joint  $J \blacktriangleright P$ , and relation  $\Rightarrow$  is  $\equiv \cup \rightarrow$ .

$$\text{NIL: } \mid\!-\! 0 \equiv \mid\!-\!$$

$$\text{PAR: } \mid\!-\! P_1 \parallel P_2 \equiv \mid\!-\! P_1, P_2$$

$$\text{JOIN: } \mid\!-\! a.(J_1 \& J_2) \equiv \mid\!-\! a.J_1, a.J_2$$

$$\text{OBJ: } \mid\!-\! \text{obj } a = D \text{ in } P \equiv a = D \mid\!-\! P$$

$$\text{REDUCTION: } a = [J \blacktriangleright P] \mid\!-\! J\sigma \rightarrow a = [J \blacktriangleright P] \mid\!-\! P\sigma$$

$$\text{COMPOSITION: } \frac{\mathcal{D} \mid\!-\! \mathcal{P}_1 \Rightarrow \mathcal{D} \mid\!-\! \mathcal{P}_2}{\mathcal{D}_0, \mathcal{D} \mid\!-\! \mathcal{P}_0, \mathcal{P}_1 \Rightarrow \mathcal{D}_0, \mathcal{D} \mid\!-\! \mathcal{P}_0, \mathcal{P}_2}$$

$$\text{COMPOS-OBJ: } \mid\!-\! P \equiv a = D \mid\!-\! \mathcal{P}, a \notin \text{fn}(D_0) \cup \text{fn}(\mathcal{P}_0)$$

$$\frac{\mathcal{D}_0 \mid\!-\! P, \mathcal{P}_0 \equiv \mathcal{D}_0, a = D \mid\!-\! \mathcal{P}_0, \mathcal{P}}{\mathcal{D}_0 \mid\!-\! P, \mathcal{P}_0 \equiv \mathcal{D}_0, a = D \mid\!-\! \mathcal{P}_0, \mathcal{P}}$$

where  $\text{fn}(P) = \cup_{P \in \mathcal{P}} \text{fn}(P)$ ,  $\text{fn}(D) = \cup_{a=D \in \mathcal{D}} (\{a\} \cup \text{fn}(D))$ .

Under  $R^*$  as a notation for reflexive, transitive closure of binary relation  $R$ , we have the following well-known propositions, which connect abstract operational semantics with “chemical” semantics (always will be understood from the context in which meaning a notation  $\equiv$  is used – either as structural congruence relation or as structural rule over configurations; similarly for  $\rightarrow$ ).

**Proposition.** *Let  $P_1, P_2$  are processes, and  $C_1, C_2$  are configurations (in  $\mathcal{D} \mid\!-\! \mathcal{P}$  form).*

*Then*

- 1)  $P_1 \equiv P_2$  iff  $\mid\!-\! P_1 \equiv^* \mid\!-\! P_2$
- 2)  $P_1 \rightarrow P_2$  iff  $\mid\!-\! P_1 \equiv^* \rightarrow \equiv^* \mid\!-\! P_2$ .

## 4.2 Distributed object calculus

Distributed object calculus is resulted from concurrent object calculus by addition of abstractions for specific constructs of MC# language: movable methods, channels and handlers.

$C ::= c(\mathbf{x})$	channel
$  C_1 \& C_2$	channel joint
$H ::= h(\mathbf{x})$	handler
$J ::= H \& C$	joint
$D ::= n(\mathbf{x}) : P$	named process
$  C \blacktriangleright P$	
$  J \blacktriangleright P$	processes with joints
$  D_1, D_2$	composite definition
$P ::= 0$	empty process
$  a.n(\mathbf{u})$	local method call
$  a.n(\mathbf{u}) \rightarrow s$	movable method call
$  a.C$	local message sending
$  C$	remote message sending
$  a.h(\mathbf{u})$	local handler call
$  h(\mathbf{u})$	remote method call
$  P_1 \parallel P_2$	concurrent composition
$  \text{obj } a = D \text{ in } P$	process with local object definition

Since names of channels and handlers can be passed as arguments to some methods (in particular, to movable methods) separately from the objects where they defined (see Section 2.2), then (remote) calls  $C$  and  $h(\mathbf{u})$  are redirected to site where mentioned object resides ( see rules CHANNEL-CALL and JOIN-CALL2) below.

We missed a construct of handler returning a value in the above calculus again to simplify the entire presentation.

We define an operational semantics of distributed object calculus by functioning rules of “chemical abstract machine”.

As before, these rules are partition on two classes: structural rules  $\equiv$  and reduction rules  $\rightarrow$ . The rules are applied to named configurations and parallel compositions of such configurations, where a name  $s$  of configuration is a name of site where computations are run.

The syntax of configurations is the following

$$A ::= s [ \mathcal{D} \mid \text{--} P ] \quad \text{site with definitions and processes}$$

$$\mid A_1 \parallel A_2 \quad \text{parallel composition}$$

The first four rules of operational semantics for distributed object calculus are similar to corresponding rules for concurrent object calculus:

$$\text{NIL} : s [ \text{--} 0 ] \equiv s [ \text{--} ]$$

$$\text{PAR} : s [ \text{--} P_1 \parallel P_2 ] \equiv [ \text{--} P_1, P_2 ]$$

$$\text{JOIN-CHAIN} : s [ \text{--} a.(C_1 \& C_2) ] \equiv s [ \text{--} a.C_1, a.C_2 ]$$

$$\text{OBJ} : s [ \text{--} \text{obj } a = D \text{ in } P ] \equiv s [ a = D \text{ --} P ]$$

A remote call of handler in location where it defined is equivalent to it local call:

REMOTE-HANDLER:

$$s [ a = [ h(x) \& C \blacktriangleright P ] \text{ --} [ h(x \sigma) ] ] \equiv$$

$$s [ a = [ h(x) \& C \blacktriangleright P ] \text{ --} a.h(x \sigma) ]$$

In the rule REMOTE-HANDL and from now on, a notation  $a = [ D ]$  is an abbreviation for object definition where one of the components is  $D$ .

The reduction rules define local and movable method calls, sending of channel messages (to site where the corresponding channel has been defined) and channel and handler joints triggering (in location where they have been defined):

LOCAL-CALL:

$$s [ a = [ n(x) : P ] \text{ --} [ a.n(x \sigma) ] ] \rightarrow$$

$$s [ a = [ n(x) : P ] \text{ --} P \sigma ]$$

MOVE:

$$s1 [ a = [ n(x) : P ] \text{ --} [ a.n(x \sigma) \rightarrow s2 ] ] \parallel s2 [ \text{--} ] \rightarrow$$

$$s1 [ a = [ n(x) : P ] \text{ --} ] \parallel s2 [ ( a = [ n(x) : P ] ) \theta ] \text{ --} a.\theta.n(x \sigma),$$

where  $\theta$  is a substitution giving globally (for all named configurations) fresh names for object  $a$  along with its channels and handlers.

CHANNEL-CALL:

$$s1 [ \text{--} c(x \sigma) ] \parallel s2 [ a = D \text{ --} ] \rightarrow$$

$$s1 [ \text{--} ] \parallel s2 [ a = D \text{ --} a.c(x \sigma) ],$$

if there is a component  $C \blacktriangleright P$  or  $J \blacktriangleright P$  in definition  $D$ , such that a channel name  $C$  is occurred in  $C$  or  $J$ , correspondingly.

JOIN-CALL1:  $s [ a = [ C \blacktriangleright P ] \text{ --} a.C \sigma ] \rightarrow$

$$s [ a = [ C \blacktriangleright P ] \text{ --} P \sigma ],$$

where  $\sigma$  is a substitution replacing formal parameters of channels in joint by actual arguments.

JOIN-CALL2:

$$s1 [ \text{--} h(x \sigma_1) ] \parallel s2 [ a = [ h(x) \& C \blacktriangleright P ] \text{ --} a.C \sigma_2 ] \rightarrow$$

$$s1 [ \text{--} ] \parallel s2 [ a = [ h(x) \& C \blacktriangleright P ] \text{ --} P(\sigma_1, \sigma_2) ],$$

where expression  $P(\sigma_1, \sigma_2)$  denotes simultaneous application of substitutions  $\sigma_1$  and  $\sigma_2$  to  $P$  under condition that  $\text{domain}(\sigma_1) \cap \text{domain}(\sigma_2) = \emptyset$

(recall that in a joint  $h(x) \& C_1(y_1) \& \dots \& C_n(y_n)$  tuples of variables are mutually not intersected).

In composition rules, as usually,  $\Rightarrow = \equiv \cup \rightarrow$ .

$$\text{COMPOS1} : \frac{s [ \mathcal{D} \text{ --} P_1 ] \Rightarrow s [ \mathcal{D} \text{ --} P_2 ]}{s [ \mathcal{D}_0, \mathcal{D} \text{ --} P_0, P_1 ] \Rightarrow s [ \mathcal{D}_0, \mathcal{D} \text{ --} P_0, P_2 ]}$$

$$\text{COMPOS2} : \frac{s1 [ \mathcal{D}_1 \text{ --} P_1 ] \Rightarrow s [ \mathcal{D}_1' \text{ --} P_1' ]}{s1 [ \mathcal{D}_1 \text{ --} P_1 ] \parallel s2 [ \mathcal{D}_2 \text{ --} P_2 ] \Rightarrow s1 [ \mathcal{D}_1' \text{ --} P_1' ] \parallel s2 [ \mathcal{D}_2 \text{ --} P_2 ]}$$

$$\text{COMPOS3} : s [ \text{--} P ] \equiv s [ a = D \text{ --} P ], a \notin \text{fn}(\mathcal{D}_0) \cup \text{fn}(P_0)$$

$$\frac{s [ \mathcal{D}_0 \text{ --} P, P_0 ] \Rightarrow s [ \mathcal{D}_0, a = D \text{ --} P_0, P ]$$

Let  $P$  be a process and  $S(P) = \{ s_1, \dots, s_2 \}$  be asset of site names, occurring in  $P$ . Then the following is an initial configuration to compute  $P$ :

$$\text{root} [ \text{--} P ] \parallel s1 [ \text{--} ] \parallel \dots \parallel \text{sn} [ \text{--} ]$$

An example of reduction of process

$$\text{obj } a = n(y) : y(), c() \blacktriangleright 0 \text{ in } a.n(c) \rightarrow s$$

given below:

$$\text{root} [ \text{--} \text{obj } a = n(y) : y(), c() \blacktriangleright 0 \text{ in } a.n(c) \rightarrow s ]$$

$$\parallel s [ \text{--} ] \Rightarrow^*$$

$$\text{root} [ a = n(y) : y(), c() \blacktriangleright 0 \text{ --} a.n(c) \rightarrow s ]$$

$$\parallel s [ \text{--} ] \Rightarrow^*$$

$$\text{root} [ a = n(y) : y(), c() \blacktriangleright 0 \text{ --} ] \parallel$$

$$s [ a' = n(y) : y(), c'() \blacktriangleright 0 \text{ --} a'.n(c) ] \Rightarrow^*$$

$$\text{root} [ a = n(y) : y(), c() \blacktriangleright 0 \text{ --} ] \parallel$$

$$s [ a' = n(y) : y(), c'() \blacktriangleright 0 \text{ --} c() ] \Rightarrow^*$$

$$\text{root} [ a = n(y) : y(), c() \blacktriangleright 0 \text{ --} a.c() ] \parallel$$

$$\begin{aligned}
& s [ a' = n ( y ) : y(), c' () \triangleright 0 | \text{--} ] \Rightarrow^* \\
& \text{root} [ a = n ( y ) : y(), c () \triangleright 0 | \text{--} 0 ] \parallel \\
& s [ a' = n ( y ) : y(), c' () \triangleright 0 | \text{--} ] \Rightarrow^* \\
& \text{root} [ a = n ( y ) : y(), c () \triangleright 0 | \text{--} ] \parallel \\
& s [ a' = n ( y ) : y(), c' () \triangleright 0 | \text{--} ].
\end{aligned}$$

## 5. IMPLEMENTATION

All described above is the development and improvement of ideas from [GS03]. At now, we have an implementation which corresponds to this paper. Some model of handlers was realized in it through so called “bi-directional channels”.

An implementation of the MC# language consist of

- 1) a compiler from MC# to C#, and
- 2) Runtime-system.

Main function of compiler is to replace the movable methods calls by queries to manager of Runtime-system that schedules execution of them. Translating of chords is conducted in the same way as in Polyphonic C# [BCF04].

The main functional parts of the Runtime-system are

- 1) Resource Manager, which is running on the frontend and realizes a centralized resource management (scheduling of movable methods calls execution), and
- 2) Worknode, a process running on each working node of cluster.

Main purpose of Worknode Process is to accept movable methods and to run their in separate threads with preliminary deserialization of main object and arguments for given method. A Worknode process have a Communicator as it compound part – a process responsible for receiving and delivering channel messages intended for objects located on given node. For this, all objects having channels are registered in the special table located on each node. Thus we have the following channel message format:

< machine address, port, object number, channel, message content >

The compiler and Runtime-system are running under Linux with Mono .NET implementation ([www.mono-project.com](http://www.mono-project.com)).

By way of experiments in MC#, we have written a big number of parallel programs: calculation of Mandelbrot set (fractals), rendering of complex images by ray tracing method, movement of body in contrary air flow (aerodynamics task), search in Internet using Google Web-service and others. In the

above tasks we used up to 64 processors. For each of the mentioned algorithms, we got an easy readable and compact code and satisfactory results concerning to efficiency of parallelizing.

## 6. CONCLUSIONS

An extension of C# language by high-level features for concurrent **distributed** programming based on local concurrent constructs of Polyphonic C#/C $\omega$  languages is presented. It can be considered as a universal language for industrial programming aimed to development of complex concurrent, distributed software applications for using on clusters and Grid-architectures.

As a theoretical base of this language, we have designed a distributed object calculus along with the formal semantics for it.

Also we built an actual implementation of MC# language (<http://u.pereslavl.ru/~vadim/MCSharp>) for running on Linux-clusters under .NET (Mono) platform.

Further theoretical work will be to develop a distributed object calculus with types. Development of typing rules for MC# expressions will provide a possibility to make more deeper static checking and optimization of program at compiling phase.

Directions of our practical work are to implement MC# language in full along with ideas presented above. Also we are going to build more effective Runtime-system with decentralized scheduling of movable methods calls and to implement “smart” serialization/deserialization algorithms.

## 7. REFERENCES

- [BCF04] Benton, N., Cardelli L., Fournet C. Modern Concurrency Abstractions for C#. ACM Transactions on Programming Languages and Systems, Vol.26, No. 5, September 2004, pp. 769-804.
- [BMS] Bierman, G., Meijer, E., Schulte, W. The essence of data access in C $\omega$ . (Submitted).
- [FG02] Fournet, C., Gonthier, G. The join calculus: a language for distributed mobile programming. – In Proc. Applied Semantics Summer School, Sept.2000. LNCS, Vol.2395, Springer, pp.268-332.
- [FLMR03] Fournet, C., Laneve C., Maranget L., Remy D. Inheritance in the join calculus. The J. of Logic and Algebraic Programming, Vol.57, 2003, pp.23-69.
- [GS03] Guzev, V., Serdyuk, Y. Asynchronous parallel programming language based on the Microsoft .NET platform. – PaCT-2003, LNCS, Vol. 2763, Springer, pp. 236- 243