A formal basis for the MC# programming language (Extended Abstract)

Yury Serdyuk

Program Systems Institute of Russian Academy of Sciences,
152020 Pereslavl-Zalessky, Russia
Yury@serdyuk.botik.ru

Abstract. In this work, we introduce the main ideas underlying a new programming language, called MC#, which is intended for concurrent distributed programming based on .NET Framework. The premises and goals for the language design are presented, and its novel constructs, such as movable methods, channels and handlers, are described. We formulate the key specific features of MC# and compare it with X10 language, one of the latest similar developments from IBM. We give an example of a concurrent distributed program in MC# for computing prime numbers with Eratosthenes sieve. In the second part of the paper, we provide a formal basis for the proposed language. The starting point is the objective join-calculus, which we extend to a distributed object calculus. For the latter, we define an operational semantics in the "abstract chemical machine" style and give an example of term reduction. We conclude with a description of the future work.

1 Introduction

In this paper, we introduce a new version of MC# — a language for .NET-based concurrent distributed programming. This language is an adaptation of the basic idea of the Polyphonic C# language [1] for the case of distributed computations.

The fundamental goal of MC# is to be a high-level, powerful and convenient programming language which would allow one to create complex and at the same time reliable software systems that efficiently use the possibilities of concurrent distributed computations and are easily scalable to a given number of processors, nodes or computers.

Currently available program interfaces and libraries for organizing parallel computations, such as OpenMP [2] (for systems with shared memory) and MPI (Message Passing Interface) [3] (for systems with message passing), have been implemented for C and Fortran languages, and hence are very low-level and inadequate for modern object-oriented programming languages like C++, C# and Java and rely on the use of libraries rather than on appropriate programming language constructs.

In general, a modern high-level programming language consists of two parts:

1) basic constructs proper, and

2) collection of specialized libraries accessible through appropriate APIs (Application Programming Interfaces).

New demands on increased programmers productivity (achieved through a higher abstraction level of language constructs, among other things), as well as on reliability and security of programs, account for a tendency to transfer key concepts of most important APIs into the corresponding native constructs of programming languages.

One of the recent achievement in this direction is an introduction of an asynchronous parallel programming model within the Polyphonic C# programming language for the Microsoft .NET Framework. In turn, this model is based on the join-calculus [4] — a process calculus with a high-level message handling mechanism adequately abstracting a low-level one, which exists in the current computer systems.

The embedding of asynchronous methods and chords into Polyphonic C# [1], which is an extension of the C# language, allows one to manage without the System.Threading library, which is normally required to implement multithreaded applications in .NET. On the other hand, the introduction of new data type constructors (for streams, anonymous structures, discriminated unions and others) along with appropriate query definition tools into Cω language [5] renders unnecessary the ADO.NET data subsystem (specifically, the traditional System.Data and System.XML libraries intended to handle relational and semistructured data).

We suggest that the next step in this direction be to introduce high-level constructs for creating concurrent distributed programs into the object-oriented language, and thus to free the programmer from the need to use the System.Remoting library (and, in many cases, also the System.Threading library), which is required to develop conventional distributed applications based on .NET.

From the practical point of view, the goal pursued by the developers of MC# was to design a language for industrial concurrent distributed programming which is going to involve more and more human resources, with the oncoming age of multicore computations. This language aims to replace C and Fortran languages in this area. In this regard, our approach coincides with that taken in the development of the X10 language [6], which is oriented towards “non-uniform cluster computing”. X10 is a part of the IBM PERCS project (Productive Easy-to-use Reliable Computer Systems), which aims to deliver by 2010 new scalable systems that will provide a 10x improvement in developing productivity for parallel applications [7].

When using the MC# language, in contrast to MPI interface, one need not do explicit distribution of computational processes over cluster nodes (although this possibility is also provided by the language) — it will suffice to specify which functions (methods) can be executed concurrently. Moreover, during the execution of programs in MC#, new computational processes can be created and distributed over accessible nodes dynamically, which is impossible in MPI-programs. A possibility to dynamically create so called “activities” is also present in the X10 language, but with the explicit indication of execution location. In X10, an asynchronous activity is created by a statement **async** (P) S, where P is a place expression and S is a statement. In contrast to MC# language with a “method level” concurrency, it is possible for multiple activities to be created in-line in a single method in X10.

MC# language is an adaptation of the basic idea of the Polyphonic C# language (more precisely, the basic idea of the join calculus [4]) for the case of concurrent

distributed computations. As a matter of fact, the authors of the Polyphonic C# language presumed that asynchronous methods would be used either on a single computer or on a set of machines where they have been fixed and interact through the remote method call tools provided by the .NET Remoting library. In the case of MC#, the execution of an autonomous asynchronous method can be scheduled on a different machine selected in either of two ways: by explicit indication by the programmer (which is not a typical case) or automatically (in this case, usually a cluster node with the least workload is selected). Interaction of asynchronous methods that are executed on different machines is realized through message passing using channels and channel message handlers. In MC#, channels and handlers are defined using chords in the Polyphonic C# style.

As our theoretical contribution, we propose object calculi for concurrent and distributed computations and define a precise operational semantics for them. This semantics is a formalization for the basic algorithm of MC# Runtime-system. In particular, for the distributed object calculus we define an operational semantics in the "abstract chemical machine" style and give an example of term reduction.

The paper is organized as follows. Section 2 describes the novel constructs of the MC# language — movable methods, channels and channel message handlers. In Section 3, we demonstrate how MC# constructs can be applied to develop a concurrent distributed program for computing prime numbers by Eratosthenes sieve. Section 4 presents a formal semantics of MC# language abstracted to concurrent and distributed object calculi. Finally, we provide conclusions and directions for future work in Section 5.

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 Polyphonic C# (which became a proper part of the Cω language — and from now on we will refer only to the latter) is the introduction of so called "asynchronous" methods in addition to conventional synchronous methods. Indeed, such asynchronous methods are intended for playing two major roles in programs:

1) the role of autonomous methods implementing the concurrent parts of the basic algorithm and executed in separate threads, and

2) that of the methods intended for delivering data (possibly, with preliminary processing of it) to conventional, synchronous methods.

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

1) movable methods and

2) channels,

respectively. In Cω, auxiliary asynchronous methods used for data delivery are usually declared together with synchronous methods. In MC#, the latter are represented as another special syntactic category of **channel message handlers** (**channel handlers** or even **handlers** for short).

2.1 Movable methods

Writing a parallel program in MC# language reduces to labeling with the special keyword **movable** the methods that may be transferred to other machines for execution:

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

In MC#, movable methods are the only way to create and run the concurrent distributed processes. A consequence of the mentioned above properties of the movable methods is that:

1) method call completes almost immediately (time is spent only on transferring the needed data to the remote machine),

2) movable methods never return a result (for interaction of movable methods among them and with other parts of the program, see Section 2.2 “Channels and handlers”).

Correspondingly, by the rules of correct definition, movable methods:

- may not have a **static** modifier, and
- never use a **return** statement.

The method call has two syntactical forms:

1) object_name.method_name (arguments)

- in this case, the Runtime-system selects the execution location for a given movable method automatically, and

2) machine_name@object_name.method_name (arguments)

- in this case, the execution location is indicated by the programmer explicitly.

It is worth to note that the objects created during an MC# program execution are **static** by their nature: once created, they don’t move and remain bound to the place where they were created. In particular, it is on this place that they are registered by the Runtime-system for delivering channel messages to those objects.

The first key feature of MC# language (or, more precisely, of its semantics) is that, in general, during a movable method call, all necessary data, namely:

1) the object itself to which the given movable method belongs, and

2) arguments (both objects and scalar values) for the latter

are only **copied** (but not moved) to the remote machine. As a consequence, changes made afterwards to the copy will not affect the original object.

In particular, if a copied object has channels or handlers, they also are copied to the remote machine — they become “proxies” for the original objects (see Section 2.2).

There are two modes of parallelizing MC# programs: “functional” and “nonfunctional” (or objective), and the choice will, in the end, affect the efficiency of program execution. These modes are defined by the modifiers **functional** and **nonfunctional** in the movable method declaration (the default value is **functional**).

In the functional mode, an object for which a movable method is called, is not transferred to a remote machine (i.e., all needed data are passed to the movable method through its arguments). Conversely, by specifying the **nonfunctional** modifier, we force the object to be moved to the remote machine.

The use of MC# on cluster architectures, which typically consist of the frontend machine and the subordinate nodes, is specific in that the names for both the frontend and the node are to be specified if a movable method is being called under explicit indication of execution location:

machine_name : node_name@o.m (args)

2.2 Channels and handlers

Channels and channel message handlers are the tools to support the interaction of distributed objects. Syntactically, channels and handlers are declared using chords in the Cω style. In the following example, the channel *sendInt* for transferring single integers is defined along with the corresponding handler *getInt*:

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

In such declarations, handlers have the following general format:

modifiers **CHandler** handler_name return_type (args)

We can also declare a channel or a group of channels without a handler. In this case, we can use values being received by the channel through the global variables.

By the rules of correct definition, channels cannot have a **static** modifier, and so they are always bound to some object much in the same way as ordinary methods:

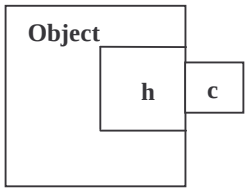

Fig. 1. An object with channel *c* and handler *h*

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

```
a.sendInt ( x ),
```

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

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

```
int  m = a.getInt ( )
```

If, by the time a handler is called, the channel is empty (i.e. if there have been no calls to this channel at all or all of the values sent through this channel before were selected during previous calls to the handler), then the call blocks. After receiving a value from the corresponding channel, the body of the chord (which may consist of arbitrary computations) runs and returns the result value to the handler.

Conversely, if a value is sent on a channel when there are no pending calls to the handler, the value is simply saved in the internal channel queue, where all the values coming with multiple sendings to this channel are accumulated. It is worth to note that separate methods (handler or channels) from the chord are typically called from different threads of which the entire concurrent distributed program consists.

Similarly to Cω, it is possible to define several channels in a single chord. This is a major tool for synchronizing the concurrent processes in MC#:

```
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 the following: the body of a chord is executed only after **all** methods declared in the chord header have been called (see the *joint-call* rule from the operational semantics definition in Section 4).

The above example illustrates the case of a single handler for multiple channels:

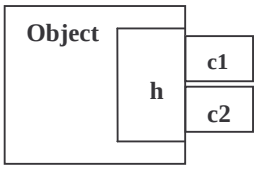

Fig. 2. An object with a single handler for multiple channels

It is also possible to declare a channel shared by several handlers:

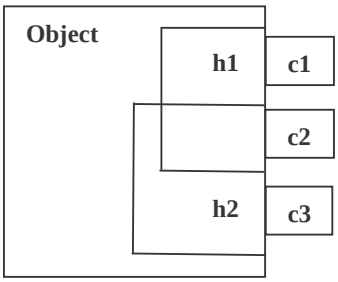

Fig. 3. An object with a "shared" channel

So, once we have values in both channels *c1*and *c2*, handler *h1* can be triggered. Similar is the case for channels *c2* and *c3* and handler *h2*. In general, all this together leads to non-determinism in program behaviour.

The second key feature of MC# language is that channels and handlers can 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 the pointers to methods or, in C# terms, to the **delegates**).

The third key feature of MC# language is that if channels or handlers were copied to a remote site (by which we mean a cluster node or an arbitrary computer in the network) autonomously or as part of some object, then they become proxy objects, or intermediaries for the original channels and handlers. This replacement is hidden from the applied programmer — he can use the passed channels and handlers (in fact, their proxy objects) on the remote site as the original ones: as usual, all actions over the proxy objects are transferred to the original channels and handlers by the Runtime-system (see the *remote-channel-call* and *remote-handler-call* rules from the operational semantics definition in Section 4). In this sense, channels and handlers are different from ordinary objects: manipulations over the latter on a remote site are not transferred to the original objects (see the first key feature of MC# language).

Fig. 4 and Fig. 5 schematically demonstrate the passing and use of channels and handlers on a remote site. The subscripts in the channel and handler names denote the original site where they were created.

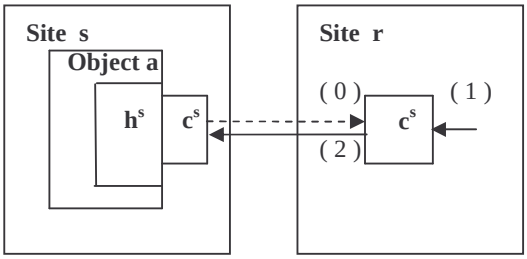

Fig. 4. Message sending by remote channel:
(0) copying of the channel to remote site,
(1) message sending by (remote) channel,
(2) message redirection to the original site.

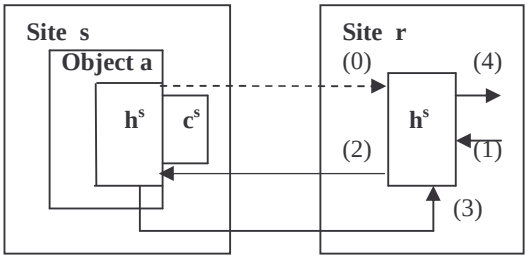

Fig. 5. Message reading from remote handler:
(0) copying of the handler to remote site,
(1) message reading from (remote) handler,
(2) reading redirection to the original site,
(3) message return from the original site, use.
(4) result message return.

It turns out that these tools are enough to organize interaction of arbitrary complexity between the concurrent distributed processes.

3 Programming in MC#

In this section, we will demonstrate the specific constructs of MC# language — movable methods, channels and handlers — and their semantic properties in informal way, on the example of real concurrent distributed program.

As an example of programming in MC#, we will build a parallel distributed program for finding prime numbers by the sieve method (also known as "Eratosthenes sieve"). Given a natural number N, we need to enumerate all primes in the interval from 2 to N. The sieving method is the following recursive procedure applied to the original list [2 , … , N]:

1) select the head of the given list and output it to the resulting list of primes;
2) construct a new list by deleting from the given list all integers that are multiples of the head of this list;
3) apply the given procedure to the newly constructed list.

The main computational subroutine, which we called *Sieve* and the recursive calls to which will be distributed over a computer network, has two arguments: the handler *getList* to read the given list of numbers it will search for primes and the channel *sendPrime* to write the resulting list of primes. The end marker in both lists is -1.

The full program text in MC# is given below. The original integer list [2 , … , N] is sent on the channel *Nats* and the resulting list of primes is received from the channel *sendPrime* by the 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.sendPrime );
  for ( int n=2; n <= N; n++ )
    E.Nats ( n );
  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 sendPrime( int p ) {
    return ( n );
 }
}
class  CSieve  {
 movable Sieve ( CHandler int() getList,
                          Channel (int) sendPrime ) {
  int  p = getList();
```

```
    sendPrime ( p );
    if  ( p != -1 )   {
     new CSieve().Sieve ( hin, sendPrime );
     filter ( p, getList, cout );
    }
   }
   CHandler hin int()  &  Channel cout ( int x ) {
    return  ( x );
   }
  void filter (int p, CHandler int() getList,
                      Channel ( int ) cfiltered )   {
    while ( ( int n = getList() ) != -1 )
      if ( n % p != 0 ) cfiltered ( n );
    cfiltered ( -1 );
  }
}
```

4 Formal semantics of MC#

To present the operational semantics of MC# in a more compact and understandable form,

- first, we define semantics only for specific constructs of MC# as movable methods, channels and handlers along with the generic constructs as concurrent composition and object declaration;
- second, we define semantics in two steps: to begin with, for concurrent (local) object calculus, and then for distributed calculus, which explicitly includes a notion of execution location — a site.

4.1 Concurrent object calculus

Let $\mathcal{N}$ be a set of names, where *x,y,* … are informally identified as *variable* names, *a,b* … as *object* names, and *m,* … as *method* names. We will denote tuples (finite lists) of names by $\boldsymbol{x}$, $\boldsymbol{y}$, etc. We use symbols σ and θ (possibly, with subscripts) to denote the mappings from the finite subsets of $\mathcal{N}$ to finite subsets of $\mathcal{N}$, and will refer to such mappings as substitutions. Suppose that if $\sigma : \mathcal{N}_1 \to \mathcal{N}_2$ is a substitution, then *domain* $(\sigma) = \mathcal{N}_1$ and *range* $(\sigma) = \mathcal{N}_2$.

The syntax of the concurrent object calculus, as a first approximation of the core of MC#, is given below. This calculus is a slightly simplified version of the **objective** join-calculus [8]:

J	:: =	$m\langle \boldsymbol{x} \rangle$	*message*
	|	$J_1 \,\&\, J_2$	*joint*
D	:: =	$\bot$	*empty definition*
	|	$J \triangleright P$	*process with joint*
	|	D_1 , D_2	*composite definition*

P :: = 0 *empty process*
| a.J *message sending*
| P_1 || P_2 *concurrent composition*
| obj a = D in P *process with the local object definition*

The concurrent object calculus consists of three syntactical categories of joints J, definitions D and processes P. In this calculus, for more compact representation of operational semantics, we generalize a message sending operator $a.m\langle \boldsymbol{x} \rangle\sigma$, where σ is a substitution replacing formal parameters of the message by actual values, to a.J — an operator for simultaneous sending of multiple messages. Here, it is worth to note that by the rules of correct definition, formal parameters lists $\boldsymbol{x_i}$ and $\boldsymbol{y_j}$ ($i \neq j$) in a joint

$$m_1 \langle \boldsymbol{x_1} \rangle \ \& \ ... \ \& \ m_n \langle \boldsymbol{x_n} \rangle$$

do not intersect pairwise.

Furthermore, in the given calculus we explicitly define a concurrent composition operator ||, whose presence in MC# is only implicit: each message sending and movable method call in it results in creating a concurrent process. Moreover, our concurrent object calculus has no methods at all — they will appear in the distributed object calculus (see Section 4.2).

Typically, there are two variants of operational semantics definitions for general concurrent calculi:

1) an abstract operational semantics, which uses a structural congruence relation, and
2) a "chemical abstract machine" style semantics [9], which is closer to actual implementations.

Below, we give both variants of semantics for the concurrent object calculus along with their simple relationships.

4.1.1 Abstract operational semantics

Let σ be a substitution. An abstract operational semantics for concurrent object calculus is based on a reduction relation $\rightarrow$ on processes defined by one axiom and three inference rules:

reduction: obj a = J ▸ P,D in a.Jσ || Q $\rightarrow$ obj a = J ▸ P,D in Pσ || Q for some σ

par: $$\frac{P \rightarrow P'}{P \,||\, Q \rightarrow P' \,||\, Q}$$

obj: $$\frac{P \rightarrow P'}{\text{obj a=D in } P \rightarrow \text{obj a=D in } P'}$$

struct: $$\frac{P \equiv Q \ \rightarrow \ Q' \equiv P'}{P \rightarrow P'}$$

Here, as usual, $\equiv$ denotes a structural congruence relation used to rearrange the term structure, which, in general, is needed to apply the axiom and the rules given above. In view of our goals, the structural congruence relation is defined as a smallest equivalence relation $\equiv$ on processes and definitions, satisfying the following axioms (α-convertibility is treated as in a λ-calculus, and free and bound names are defined below):

1. $P \equiv P'$, if P and P' are α-convertible
2. $D \equiv D'$, if D and D' are α-convertible
3. $P \,||\, Q \equiv P' \,||\, Q'$, if $P \equiv P'$ and $Q \equiv Q'$
4. $D_1 , D_2 \equiv D_1' , D_2'$, if $D_1 \equiv D_1'$ and $D_2 \equiv D_2'$
5. $\underline{\text{obj}}\ a = D\ \underline{\text{in}}\ P \equiv \underline{\text{obj}}\ a = D'\ \underline{\text{in}}\ P'$, if $D \equiv D'$ and $P \equiv P'$
6. $P \,||\, 0 \equiv P$
7. $D , \bot \equiv D$
8. $P_1 \,||\, P_2 \equiv P_2 \,||\, P_1$
9. $D_1 , D_2 \equiv D_2 , D_1$
10. $P \,||\, (Q \,||\, R) \equiv (P \,||\, Q) \,||\, R$
11. $D_1 , (D_2 , D_3) \equiv (D_1 , D_2), D_3$
12. $(\underline{\text{obj}}\ a = D\ \underline{\text{in}}\ P) \,||\, Q \equiv \underline{\text{obj}}\ a = D\ \underline{\text{in}}\ (P \,||\, Q)$, if $a \notin fn (Q)$
13. $P \equiv Q \ \Rightarrow\ \boldsymbol{C} [P] \equiv \boldsymbol{C} [Q]$ for any context $\boldsymbol{C}$
    (we do not define a notion of context formally here).

Here, *fn (Q)* is a set of free names of a process *Q*. A name *n* is *free* in process *Q* if it is not *bound* in this process. In turn, a set *bn (Q)* of bound names of process *Q* is defined as

1. $bn (0) = \varnothing$
2. $bn (a.J) = \varnothing$
3. $bn (P_1 \,||\, P_2) = bn (P_1) \cup bn (P_2)$
4. $bn (\underline{\text{obj}}\ a = D\ \underline{\text{in}}\ P) = \{ a \} \cup bn (D) \cup bn (P)$
5. $bn (\bot) = \varnothing$
6. $bn (J \triangleright P) = bn (J) \cup bn (P)$
7. $bn (m \langle x_1, \ldots , x_n \rangle) = \{ x_1, \ldots , x_n \}$
8. $bn (J_1 , J_2) = bn (J_1) \cup bn (J_2)$

4.1.2 Operational semantics in “chemical abstract machine” style

An operational semantics defined as the working rules of a chemical abstract machine [9] is closer to actual implementations of concurrent and/or distributed languages. In our case, it is given as a set of rewriting rules applied to process and object configurations. A configuration

$$\mathcal{D} \vdash \mathcal{P}$$

consists of a set $\mathcal{D}$ of object definitions and a multiset $\mathcal{P}$ of (concurrently running) processes. A configuration is a “solution” in terms of chemical abstract machine. The “chemical reactions” happening in a solution are given by the rewriting rules of two kinds: structural rules $\equiv$ making term rearrangements and reduction rules $\rightarrow$ representing computation steps.

According to the chemical abstract machine style definition, the rules for rearrangement and computation steps show only the processes and definitions that are directly involved in the step. The rest of the solution, which remains unchanged, is implicit.

Below, notation $a = \{ J \triangleright P \}$ is an abbreviation for an object declaration which includes, among others, a definition $J \triangleright P$.

1. *nil*: $\vdash 0 \equiv \vdash$
2. *par*: $\vdash P_1 \,||\, P_2 \equiv \vdash P_1 , P_2$
3. *joint*: $\vdash a.J_1 \,\&\, J_2 \equiv \vdash a.J_1 , a.J_2$
4. *obj*: $\vdash \underline{\text{obj}}\ a = D\ \underline{\text{in}}\ P \equiv (\ a = D\ )\sigma \vdash P\sigma$,
   where σ is a substitution with *domain* $(\sigma) = \{a\}$, which gives a fresh name to an object *a* in an enclosed configuration.
5. *reduction*: $a = \{\ J \triangleright P\ \} \vdash a.J\sigma \rightarrow a = \{\ J \triangleright P\ \} \vdash P\sigma$,
   where a substitution σ gives actual values to the message parameters in a joint J.

Let R^* be a reflexive, transitive closure of a binary relation *R*. Then the following well-known propositions (see [4]), linking an abstract operational semantics and a "chemical" semantics hold true (it is always clear from the context in which meaning notation $\equiv$ is used: either as a notation for a structural congruence relation over processes or a notation for structural rules over configurations; the same holds for $\rightarrow$).

Proposition. *Let P_1, P_2 be processes of a concurrent object calculus. Then*

1) $P_1 \equiv P_2$ *iff* $\vdash P_1 \stackrel{*}{\equiv} \vdash P_2$,

2) $P_1 \rightarrow P_2$ *iff* $\vdash P_1 \stackrel{*}{\equiv} \rightarrow \stackrel{*}{\equiv} \vdash P_2$.

4.2 Distributed object calculus

A distributed object calculus results from adding to a concurrent object calculus abstractions for specific constructs of MC# language such as movable methods, channels and handlers. In particular, in the given calculus, movable method calls with an implicit indication of execution location are replaced by those with the explicit indication of such. So, this calculus can be considered as a formal model of language with remote procedure calls (RPC).

Now, in a set $\boldsymbol{N}$ of (simple) names we informally identify *variable* names *x,y,z, …*, *object* names *a,b, … ,* object *method* names *m, …*, *channel* names *c, … , handler* names *h, … ,* and *site* names *s, r, … .* In general, names can have subscripts and superscripts. In addition to simple names, we will use composite names as *a.m*, *a.c*, *a.h* to denote object components such as methods, channels and handlers, respectively. In what follows, the word "name" means either a simple or a composite name. To denote tuples (finite lists) of expressions, we will use a notation $\boldsymbol{E}$.

The syntax of the distributed object calculus is given below:

C :: =		*channels*
	$c\langle \boldsymbol{x} \rangle$	*channel*
	$\mid C_1 \,\&\, C_2$	*channel joint*
H :: =	h ($\boldsymbol{y}$)	*handler*
J :: =	H & C	*general joint*
D :: =		*object method definition*
	m ($\boldsymbol{x}$) : P	*named process*
	$\mid$ C $\triangleright$ P	*process with channel joint*
	$\mid$ J $\triangleright$ *return* E	*expression with joint*
	$\mid D_1 , D_2$	*composite definition*

O :: =		*object definition*
	a = D	*single object definition*
	$\mid$ O_1 ; O_2	*composite definition*
E :: =		*expression*
	u	*name*
	$\mid$ h (***E***)	*handler call*
P :: =		*process*
	0	*empty process*
	$\mid$ a.m (***E***)	*method call*
	$\mid$ a.m (***E***) $\rightarrow$ s	*movable method call*
	$\mid$ a.c $\langle$ ***E*** $\rangle$	*message sending*
	$\mid$ $P_1 \mid\mid P_2$	*concurrent composition*
	$\mid$ obj a = D in P	*local object definition*

We call the expression of the form $c^s\langle \boldsymbol{x} \rangle$ *a channel localized relative to site* s. Correspondingly, D^s denotes an object method definition, in which all channels and handlers are localized relative to site *s*. Finally, let O be an object definition. Then O^s is a set of definitions a = D^s for each a = D from O. Informally, objects (and, correspondingly, their channels and handlers) are localized relative to site *s* during their initial creation on that site.

Below, we define an operational semantics of the distributed object calculus in a "chemical abstract machine" style. As before, the rewriting rules for chemical machine operation are divided into two classes: structural rules $\equiv$ and reduction rules $\rightarrow$. They are applied to the configurations whose syntax is as follows:

A :: =		*site configuration*
	$s[\ \mathcal{D} \vdash \mathcal{P}\ ]$	*site*
	$\mid A_1 \mid\mid A_2$	*concurrent composition of sites*

As usual, each rewriting rule shows only those sites and, within the sites, only those processes and definitions that are directly involved in the computation step. The other sites with their processes and definitions remain unchanged.

Let σ and θ be some substitutions such that *domain* $(\sigma) \cap$ *domain* $(\theta) = \varnothing$. Then, $\sigma \circ \theta$ denotes a new substitution with the *domain* $(\sigma \circ \theta)$ = *domain* $(\sigma) \cup$ *domain* (θ) such that for any name *u*

$u(\sigma \circ \theta) = \sigma(u)$, if $u \in$ *domain* (σ),
$u(\sigma \circ \theta) = \theta(u)$, if $u \in$ *domain* (θ),
$u(\sigma \circ \theta) = u$, otherwise.

Also, $\boldsymbol{u} \Rightarrow \boldsymbol{v}$ denotes a substitution that replaces each u_i from $\boldsymbol{u}$ with v_i from $\boldsymbol{v}$, supposing that the lengths of $\boldsymbol{u}$ and $\boldsymbol{v}$ are equal.

1. *nil*: $s[\ \vdash 0\ ] \equiv s[\ \vdash\ ]$
2. *par*: $s[\ \vdash P_1 \mid\mid P_2\ ] \equiv s[\ \vdash P_1, P_2\ ]$
3. *channel-joint*: $s[\ \vdash a.C_1 \,\&\, C_2\ ] \equiv s[\ \vdash a.C_1, a.C_2\ ]$
4. *obj*: $s[\ \vdash$ obj O in P$] \equiv s[\ O^s\sigma \vdash P\sigma]$,
   where σ is a substitution giving fresh (relative to site *s*) names to the objects occurring in definition O.
5. *local-call*: $s[\ a = \{m(\boldsymbol{x}): P\} \vdash a.m(\boldsymbol{u})\ ] \rightarrow s[\ a = \{m(\boldsymbol{x}): P\} \vdash P(\boldsymbol{x} \Rightarrow \boldsymbol{u})]$
6. *movable-call*: $s[O^s \vdash a.m(\boldsymbol{u}) \rightarrow r] \mid\mid r[\ \vdash\ ] \equiv s[O^s \vdash\ ] \mid\mid r[O^s\sigma \vdash (a.m(\boldsymbol{u}))\sigma]$,

where σ is a substitution giving fresh (relative to site *r*) names to the objects occurring in definition O.

7. *channel-call*: $s\,[\,a = \{\,C^s \triangleright P\,\} \mid a.C\sigma\,] \to s\,[\,a = \{\,C^s \triangleright P\,\} \mid P\sigma\,]$
for some substitution σ that gives actual values to message parameters in the channel joint C.
8. *joint-call*: $s\,[a = \{h^s(\boldsymbol{x}) \,\&\, C^s \triangleright \underline{return}\ E\} \mid P\,\{h(\boldsymbol{u})\},\ a.C\sigma] \to$
$s\,[a = \{h^s(\boldsymbol{x}) \,\&\, C^s \triangleright \underline{return}\ E\} \mid P\,\{E\,(\boldsymbol{x} \Rightarrow \boldsymbol{u} \circ \sigma\,/\,h(\boldsymbol{u})\}]$
for some substitution σ that gives actual values to formal parameters.
9. *remote-channel-call*: $s\,[\,a = D^s \mid]\ ||\ r\,[\,a = D^s \mid a.c\langle \boldsymbol{u}\rangle\,] \equiv$
$s\,[\,a = D^s \mid a.c\langle \boldsymbol{u}\rangle\,]\ ||\ r\,[\,a = D^s \mid]$,
if $r \neq s$, and a definition D contains a joint C ▷ P or a joint H & C ▷ *return* E, where C contains *c*.
10. *remote-handler-call*: $s\,[\,a = D^s \mid]\ ||\ r\,[\,a = D^s \mid P\,\{\,a.h(\boldsymbol{u})\,\}\,] \equiv$
$s\,[a = D^s; p = F^r \mid p.c\langle\, a.h(\boldsymbol{u})\rangle]\ ||\ r\,[a = D^s; p = F^r \mid P\{p.h() / a.h(\boldsymbol{u})\}\,]$
if $r \neq s$, and D = h (***x***) & C ▷ *return* E for some C and E, and *p* is a new object with a definition F = h() & c⟨y⟩ ▷ *return* y on sites *s* and *r*.

The first four rules of operational semantics for the distributed object calculus are similar to the corresponding rules for the concurrent object calculus. In the rule *movable-call*, relocation of definition O^s from site *s* to site *r*, in fact, represents a sequence of actions "serialization — transferring to remote site — deserialization" over the objects participating in a movable method call. In the rules *joint-call* and *remote-handler-call*, P { E } denotes a process P with an occurrence of expression E, and P $\{E_2 / E_1\}$ denotes a process resulting from replacing expression E_1 with expression E_2 in the process P. In the rule *remote-handler-call*, an object *p* plays a role of a proxy object readdressing a value computed by handler *h* on site *s* to site *r*.

Let P be a process and R (P) = { $r_1, \ldots, r_n$ } be a set of site names occurring in P. Then a configuration

$$s\,[\ \mid P\,]\ ||\ r_1\,[\ \mid\ ]\ ||\ \ldots\ ||\ r_n\,[\ \mid\ ]$$

is an *initial configuration* to compute the process P (for some distinguished site name s which differs from any r_i).

Below is an example of reduction for the process

$$P = \underline{obj}\ a = D_1;\ b = D_2\ \underline{in}\ a.m\,(\,a.c\,) \to r \mid\mid b.print\,\langle\, a.h\,()\,\rangle,$$

where

$$D_1 = m\,(\,x\,) : x\,\langle\, u\,\rangle,\ h\,(\,) \,\&\, c\,\langle\, y\,\rangle \triangleright \underline{return}\ y,$$
$$D_2 = print\,\langle\, x\,\rangle \triangleright 0.$$

Then for an initial configuration

$$s\,[\ \mid P\,]\ ||\ r\,[\ \mid\ ]$$

we have the following sequence of reductions:

$s\,[\mid \underline{obj}\ a = D_1;\ b = D_2\ \underline{in}\ a.m\,(\,a.c\,) \to r \mid\mid b.print\,\langle\, a.h\,()\,\rangle\,]\ ||\ r\,[\ \mid\ ]\ \equiv$
$s\,[\,a = D_1^s;\ b = D_2^s \mid a.m\,(\,a.c\,) \to r \mid\mid b.print\,\langle\, a.h\,()\,\rangle\,]\ ||\ r\,[\ \mid\ ]\ \equiv$
$s\,[\,a = D_1^s;\ b = D_2^s \mid a.m\,(\,a.c\,) \to r\ ,\ b.print\,\langle\, a.h\,()\,\rangle\,]\ ||\ r\,[\ \mid\ ]\ \equiv$
$s\,[\,a = D_1^s;\ b = D_2^s \mid b.print\,\langle\, a.h\,()\,\rangle\,]\ ||\ r\,[\ a = D_1^s;\ b = D_2^s \mid a.m\,(\,a.c\,)\,] \to$
$s\,[\,a = D_1^s;\ b = D_2^s \mid b.print\,\langle\, a.h\,()\,\rangle\,]\ ||\ r\,[\ a = D_1^s;\ b = D_2^s \mid a.c\,\langle\, u\,\rangle\,] \equiv$
$s\,[\,a = D_1^s;\ b = D_2^s \mid b.print\,\langle\, a.h\,()\,\rangle, a.c\,\langle\, u\,\rangle\,]\ ||\ r\,[\ a = D_1^s;\ b = D_2^s \mid\ ] \to$
$s\,[\,a = D_1^s;\ b = D_2^s \mid b.print\,\langle\, u\,\rangle\,]\ ||\ r\,[\ a = D_1^s;\ b = D_2^s \mid\ ] \to$

$s\,[\,a = D_1^s;\, b = D_2^s \,\vdash\, 0\,]\;||\;r\,[\,a = D_1^s;\, b = D_2^s \,\vdash\,]\;\equiv$
$s\,[\,a = D_1^s;\, b = D_2^s \,\vdash\,]\;||\;r\,[\,a = D_1^s;\, b = D_2^s \,\vdash\,]$

5 Conclusions

This work presents an extension of C# language with high-level features for concurrent distributed programming based on the asynchronous programming model of Polyphonic C#. It can be considered as a general-purpose language for practical industrial programming, which is oriented towards creating complex parallel software systems intended to run on cluster architectures.

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

Also we built a prototype implementation of MC# language for Linux cluster and a network of Windows machines [10]. (The MC# project site is at: http://u.pereslavl.ru/~vadim/MCSharp)

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

Our future practical work will focus on implementing the MC# language in full accordance with the ideas put forward in the paper. Also, a version of MC# programming system for metacluster computations is under development.

References

1. Benton, N., Cardelli, L., Fournet, C.: Modern Concurrency Abstractions for C#. ACM Trans. Prog. Lang. Sys. 26, 5 (2004) 769–804
2. OpenMP specifications: www.openmp.org/specs
3. Message Passing Interface: www-unix.mcs.anl.gov/mpi/
4. Fournet, C., Gonthier, G.: The Join Calculus: a Language for Distributed Mobile Programming. Proc. Applied Semantics Summer School, 2000. Lecture Notes in Computer Science, Vol. 2395. Springer-Verlag, Berlin Heidelberg New York (2000) 268-332
5. Bierman, G., Meijer, E., Schulte, W.: The Essence of Data Access in Cω. ECOOP 2005. Lecture Notes in Computer Science, Vol. 3586. Springer-Verlag, Berlin Heidelberg New York (2005) 287-311
6. Saraswat, V., Jagadeesan, R.: Concurrent Clustered Programming. CONCUR 2005. Lecture Notes in Computer Science, Vol. 3653. Springer-Verlag, Berlin Heidelberg New York (2005) 353-367
7. Charles, P., Grothoff, C., Donawa, C., Ebciouglu, K., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing. Proc. ACM 2005 OOPSLA Conf., Onward! Track, October 2005, 519-538
8. Fournet, C., Laneve, C., Maranget, L., Remy, D.: Inheritance in the Join Calculus. J. Logic and Algeb. Prog. 57 (2003) 23-69
9. Berry, G., Boudol G.: The Chemical Abstract Machine. Theor. Comput. Sci. 96 (1992) 217-248

10. Guzev, V., Serdyuk, Y.: Asynchronous Parallel Programming Language Based on the Microsoft .NET Platform. PaCT 2003. Lecture Notes in Computer Science, Vol. 2763. Springer-Verlag, Berlin Heidelberg New York (2003) 236-243