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

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

**Abstract**

We introduce the main ideas underlying a new programming language, called MC#, which is intended for development of industrial concurrent distributed applications based on .NET Framework. The premises and goals for the language design are presented, its novel constructs, such as movable methods, channels and handlers, are described, and the key specific features of MC# are formulated. In the second part of the paper, we provide a formal basis for the proposed language. The starting point is the objective join-calculus of Fournet and Gonthier, which we extend first to a concurrent object calculus (C-calculus) and then to a distributed object calculus (D-calculus). For both calculi, we define an operational semantics in the "abstract chemical machine" style, and for the C-calculus we develop a type system and prove a "subject reduction" theorem. We conclude with a description of the current implementation and the directions for future work.

## 1 Introduction

In this paper, we introduce a formal basis for MC# - a language for .NET-based concurrent, distributed programming [2, 3]. Given language is an adaption of the basic idea of the Polyphonic C# language [1] for the case of distributed programming. As a matter of fact, the authors of the Polyphonic C# language presumed that asynchronous methods can be used either on local computer or on a set of machines where they have been fixed and interact through the remote method call tools. In the case of MC#, the execution of an autonomous asynchronous method can be scheduled on a different machine selected mostly automatically. Interaction of remote asynchronous methods - in MC#, they are called "movable methods" - or, equivalently, between distributed objects, is realized by message passing using other new first-class constructs of MC#, such as channels and handlers. The latter are defined using chords in the Polyphonic C# style.

## 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 one of the key feature of Polyphonic C# is the introduction of so called "asynchronous" methods in addition to conventional synchronous ones. Indeed, such asynchronous methods are intended to play the two major roles in concurrent programs:

1. to implement the basic part of the algorithm in separate threads, and
2. to deliver the data (possibly, with preliminary processing of it) to conventional, synchronous methods.

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

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

respectively. In Polyphonic C#, asynchronous methods used for data delivery are usually declared together with synchronous methods. In MC#, the latter are represented through another first-class constructs, namely, the message handlers (or simply, handlers, for short).

*Async*- and *movable* methods are the only way to create concurrent distributed processes. In contrast to *async*-methods that are running locally, *movable* metods can be transferred to other machines for execution. In particularly, **the first key feature of MC# language** 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 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 and handlers, they also are copied to the remote machine - they become "proxies" for the original ones (see the *movable-call* rule for D-calculus, Section 3.2).

Channels and handlers are the tools to support the interaction between distributed objects. Syntactically, channels and handlers are declared using chords in the Polyphonic C# style (cf. [1] and [3]). In the following example, the channel *sendInt* for transferring single integers is defined along with the corresponding handler *getInt*:

public **handler** getInt **int**() & **channel** sendInt ( **int** x ) { **return** ( x ); }

In such declarations, handlers have the following general format:

modifiers **handler** handler-name return-type (args)

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 *reduction* rule of C-calculus, and *joint-call* rule of D-calculus in Section 3). **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#** is that if channels or handlers were copied to a remote machine separately or as part of some object, then they become proxy objects, or intermediares for the original ones. As usual, all manipulations over the proxy objects are transferred to the original channels and handlers by the Runtime-system.

Many examples of programs written in MC# and demonstrating its specific constructs in work can be found in [3] and at MC# project site www.mcsharp.net.

# 3 Formal semantics of MC#

To present the operational semantics of MC# in a more compact and understandable form, we define it only for such specific constructs of MC# as async- and movable methods, channels and handlers along with generic features, such as concurrent composition and object declaration.

The main distinction of our definition of operational semantics is that, in contrast to traditional approaches [4, 5], we do not consider object as (a guarded sum of) concurrent processes. So in object-oriented settings, we adhere to more simple and natural approach when the only concurrent processes in a calculus are method calls. Given approach allows to separate "object-oriented" issues as method update, inheritance, cloning etc., from the process calculus as such.

Following this way, we define sequentially a concurrent (local) object calculus (C-calculus) and then a distributed object calculus (D-calculus), where the latter explicitly includes a notion of execution location - a site. Also, we define a type system for C-calculus and prove "a subject reduction" theorem for it.

## 3.1 C-calculus: a concurrent object calculus

Let $N$ 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 of names by $\bar{x}, \bar{y}$, etc. The syntax of the concurrent object calculus, as a first approximation of the core of MC#, is given below. This calculus is a modified version of the objective join-calculus [6].

| | | | |
|---|---|---|---|
| $J ::=$ | | $m < \bar{x} >$ | *message (or channel)* |
| | $\mid$ | $J_1 \& J_2$ | *joint* |
| $D ::=$ | | $\bot$ | *empty definition* |
| | $\mid$ | $J \triangleright P$ | *process with joint* |
| | $\mid$ | $D_1, D_2$ | *composite definition* |
| $O ::=$ | | $a = D$ | *object declaration* |
| | $\mid$ | $O_1; O_2$ | *composite declaration* |
| $P ::=$ | | $0$ | *empty process* |
| | $\mid$ | $a.J$ | *message sending* |
| | $\mid$ | $P_1 \mid\mid P_2$ | *concurrent composition* |
| | $\mid$ | $\underline{obj}\ O\ \underline{in}\ P$ | *process with objects* |

Typically, there are two variants of operational semantics definitions for concurrent calculi: 1) an abstract operational semantics, which uses a structural conguence relation, and 2) a "chemical abstract machine" style semantics, which is closer to actual implementations.

### 3.1.1 Abstract operational semantics

Let $\sigma$ be a substitution on $N$. An abstract operational semantics for C-calculus is based on reduction relation $\rightarrow$ on processes defined by one axiom and three inference rules. Here, we drop a definition of structural congruence relation $\equiv$.

| | |
|---|---|
| *reduction* : | $\underline{obj}\ a = J \triangleright P, D; O\ \underline{in}\ a.J\sigma \parallel Q \rightarrow \underline{obj}\ a = J \triangleright P, D; O\ \underline{in}\ P\sigma \parallel Q$ |
| *par* : | $\dfrac{P \rightarrow P'}{P \parallel Q \rightarrow P' \parallel Q}$ |
| *obj* : | $\dfrac{P \rightarrow P'}{\underline{obj}\ O\ \underline{in}\ P \rightarrow \underline{obj}\ O\ \underline{in}\ P'}$ |
| *struct* : | $\dfrac{P \equiv Q \rightarrow Q' \equiv P'}{P \rightarrow P'}$ |

### 3.1.2 Chemical semantics

A chemical abstract machine for C-calculus is defined as a set of rewriting rules applied to object and process configurations. A configuration ("chemical soup") $\mathcal{O} \Vdash \mathcal{P}$ consist of a set $\mathcal{O}$ of object declarations and a multi-set $\mathcal{P}$ of (concurrently running) processes. The "chemical reactions" are given by (reversible) *structural* rules $\rightleftharpoons$ and *reduction* rules $\rightarrow$ representing computation steps. In the following, a notation $a = \{\ J\ \triangleright\ P\}$ is an abbreviation for an object declaration which includes, among others, a definition $J \triangleright P$. Below we present the most typical rule only - a reduction rule:

| | |
|---|---|
| *reduction* : | $a = \{\ J\ \triangleright\ P\} \Vdash a.J\sigma\ \rightarrow\ a = \{\ J\ \triangleright\ P\} \Vdash P\sigma$<br>where a substitution $\sigma$ gives actual values to the message parameters in $J$ |

Then the following propositions, linking an abstract operational semantics and a "chemical" semantics hold true.

**Proposition.** *Let $P_1, P_2$ be processes of C-calculus. Then*

1) $P_1 \equiv P_2$ *iff* $\Vdash P_1 \equiv^* \Vdash P_2$,

2) $P_1 \rightarrow P_2$ *iff* $\Vdash P_1 \equiv^* \rightarrow \equiv^* \Vdash P_2$.

### 3.1.3 Type system

To build a type system for C-calculus, we assume given a set of basic types $b \in \mathcal{T}$. Then the syntax of types and typing environments, which associate types to names, is the following:

| | | | |
|---|---|---|---|
| $\tau ::=$ | | $b$ | *basic type* |
| | $\mid$ | $< \tau_1, ..., \tau_n >$ | *message (channel) type* |
| | $\mid$ | $\{m_1 : \tau_1, ..., m_n : \tau_n\}$ | *object type* |
| $A ::=$ | | $\emptyset$ | *empty typing environment* |
| | $\mid$ | $A\ +\ u : \tau,$ | $u \in N$ |

Further, we use four kinds of typing judgements:

| | |
|---|---|
| $A \vdash u : \tau$ | the name $u$ has type $\tau$ in $A$ |
| $A \vdash P$ | the process $P$ is well-typed in $A$ |
| $A \vdash D : B$ | the definition $D$ is well-typed in $A$ with types $B$ for its defined channel names |
| $A \vdash O : B$ | the declaration $O$ is well-typed in $A$ with types $B$ for its declared object names |

The following rules describe valid derivations for type judgements (cf. with [7]). (Here, $B_1 \oplus B_2$ is a union of $B_1$ and $B_2$, and requires $B_1$ and $B_2$ to be equal on common names. Also, in notation $a : \{\ m_i :< \bar{\tau}_i >^{i \in 1..n}\ ; \rho\}$, a symbol $\rho$ denotes a remaining part of the type of object $a$.

$$\textit{(Name)}\ \frac{}{A\ +\ u:\tau\ \vdash\ u:\tau} \qquad \textit{(Nil)}\ \frac{}{A\ \vdash\ \emptyset}$$

$$\textit{(Joined message)}\ \frac{A\vdash a:\{m_i:<\bar{\tau}_i>^{i\in 1..n}\ ;\rho\}\ \ A\vdash \bar{v_i}:\bar{\tau}_i,\ i=1..n}{A\vdash a.m_1<\bar{v_1}>\ \&\ ...\ \&\ m_n<\bar{v_n}>}$$

$$\textit{(Par)}\ \frac{A\vdash P_1\ \ A\vdash P_2}{A\vdash P_1||P_2} \qquad \textit{(Empty-Def)}\ \frac{}{A\vdash\perp::\emptyset}$$

$$\textit{(Join-Def)}\ \frac{A+\bar{u_1}:\bar{\tau}_1+\ ...\ +\bar{u_n}:\bar{\tau}_n\ \vdash P}{A\vdash x_1<\bar{u_1}>\ \&\ ...\ \&x_n<\bar{u_n}>\rhd P\ ::\ \{x_i:<\bar{\tau}_i>^{i\in 1..n}\}}$$

$$\textit{(Compose-Def)}\ \frac{A\vdash D_1::B_1\ \ A\vdash D_2::B_2}{A\vdash D_1,D_2::B_1\oplus B_2} \qquad \textit{(Object)}\ \frac{A+a:B\vdash D::B}{A+a:B\vdash a=D::\{a:B\}}$$

$$\textit{(Compose-Obj)}\ \frac{A\vdash O_1::B_1\ \ A\vdash O_2::B_2}{A\vdash O_1,O_2::B_1\oplus B_2} \qquad \textit{(Def-Obj)}\ \frac{A+B\vdash O::B\ \ A+B\vdash P}{A\vdash \underline{obj}\ O\ \underline{in}\ P}$$

To prove a subject reduction property for C-calculus, we intoduce yet another typing judgement $A\ \vdash\ \mathcal{O}\ \Vdash\ \mathcal{P}$ to state that all declarations and all processes are independently well-typed in the same typing environment $A$. Omitting here some further technical details, we can formulate our main result for C-calculus.

**Theorem.** *One-step chemical reductions preserve types.*

## 3.2 D-calculus: a distributed object calculus

A D-calculus results from adding to C-calculus abstractions for specific constructs of MC# language such as async- and movable methods, channels and handlers and a notion of execution location - a site.

Now, within a set $N$ of (simple) names we informally identify *variable* names $x, y, z, ..$, *object* names $a, b, ...$, *method* names $m, ...$, *channel* names $c, ...$, *handler* names $h, ...$, and *site* names $s, r, ....$ We use $u \in N$ to denote a name in general. In addition to simple names, we use composite names as $a.m, a.c, a.h$ to denote object components such as methods, channels and handlers, respectively.

The syntax of the D-calculus is given below:

| | | |
|---|---|---|
| $C ::=$ | | *channels* |
| | $c<\bar{x}>$ | *channel* |
| \| | $C_1\ \&\ C_2$ | *channel joint* |
| $H ::=$ | $h(\bar{y})$ | *handler* |
| $J ::=$ | $H\ \&\ C$ | *general joint* |
| $D ::=$ | | *method definition* |
| | $m(\bar{x}):P$ | *async-method* |
| \| | $C\ \rhd\ P$ | *process with channel joint* |
| \| | $J\ \rhd\ \underline{return}\ E$ | *expression with joint* |
| \| | $D_1\ ,\ D_2$ | *composite definition* |
| $O ::=$ | | *object declaration* |
| | $a\ =\ D$ | *single object declaration* |
| \| | $O_1\ ;\ O_2$ | *composite declaration* |
| $E ::=$ | | *expressions* |
| | $u$ | *name* |
| \| | $a.h(\bar{E})$ | *handler call* |
| $P ::=$ | | *processes* |
| | $\emptyset$ | *empty process* |
| \| | $a.m(\bar{E})$ | *async-method call* |
| \| | $a.m(\bar{E})\ \rightarrow\ s$ | *movable method call* |
| \| | $a.c<\bar{E}>$ | *message sending* |
| \| | $P_1\ \|\|\ P_2$ | *concurrent composition* |
| \| | $\underline{obj}\ O\ \underline{in}\ P$ | *process with (local) objects* |

For this calculus, we developed an operational semantics 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:

| | | |
|---|---|---|
| $S ::=$ | | *site configuration* |
| | $s[\ \mathcal{O} \vdash \mathcal{P}\ ]$ | *site* |
| | $S_1 \parallel S_2$ | *concurrent composition of sites* |

Below, we provide three rules of chemical semantics to give a flavour of it. A notation like $O^s$ denote an object declaration, in which all channels and handlers for all objects in $O$ are bounded (or, in other words, are localized relative) to site $s$. Informally, objects are bounded to site $s$ during their initial creation on this site (see the rule *obj* below). Also, $P\{E\}$ denotes a process with an occurence of expression $E$, and $E_1/E_2$ denotes replacement of $E_1$ with $E_2$ in corresponding term. As usual, chemical rewriting rules show only those components that are directly involved in computation steps. The other parts of chemical soup remain unchainged.

| | |
|---|---|
| $\underline{obj}:$ | $s[\vdash \underline{obj}\ O\ \underline{in}\ P\ ] \equiv s[\ O^s\sigma \vdash P\sigma\ ]$, where $\sigma$ is a substitution giving fresh (relative to site $s$ ) names to the objects in $O$. |
| $\underline{movable-call}:$ | $s[\ O^s \vdash a.m(\bar{u}) \rightarrow r\ ] \parallel r\ [\vdash\ ] \equiv s[\ O^s \vdash\ ] \parallel r\ [\ O^s\sigma \vdash (a.m(\bar{u}))\sigma\ ]$, where $\sigma$ is a substitution giving fresh (relative to site $r$ ) names to the objects in $O^s$. |
| $\underline{joint-call}:$ | $s[a = \{h^s(\bar{x})\ \&\ C^s \rhd \underline{return}\ E\} \vdash P\{h(\bar{u})\}, a.C\sigma\ ] \rightarrow$ $s[a = \{h^s(\bar{x})\ \&\ C^s \rhd \underline{return}\ E\} \vdash P\{h(\bar{u})/E(\bar{x}/\bar{u})\sigma\ \}\ ]$ for some substitution $\sigma$ that gives actual values to formal parameters of $C$. |

In fact, the above rules together with those omitted, embody in pure form the basic algorithm of Runtime-system for MC# language.

# 4 Implementation and future work

We build an implementation of MC# language that consist of a compiler from MC# to C#, and a Runtime-system. In fact, there are 3 versions of MC# programming system: for multi-core machines, for clusters a nd for metacluster/Grid-systems. By way of experiments, we have written a large series of concurrent distributed applications in MC#, such as 3D rendering, radar-tracking signals processing, molecular dynamics computations, etc. MC# language was used with success in Intel Threading Challenge contest (http://softwarecontests.intel.com/threadingchallenge). Large-scale tests were conducted on the "SKIF K-1000" ($98^{th}$ in Top500, November 2004) and "SKIF Cyberia" ($105^{th}$ in Top500, June 2007) clusters. The maximum number of processors used was 1100. MC# site is http://www.mcsharp.net.
Further theoretical effort will be to establish the properties of proposed semantics and to develop a type system for D-calculus. Also we plan to add to the language specific constructs that allow to define data (mostly, arrays) distribution between movable methods in declarative manner like in X10 language [8].

# References


[1] N. Benton, L. Cardelli, C. Fournet "Modern Concurrency Abstractions for C#". ACM Transactions on Programming Languages and Systems, vol. 26, N 5, 2004, pp. 769-804.

[2] V. Guzev, Yu. Serdyuk "Asynchronous parallel programming language based on the Microsoft .NET platform". PaCT-2003, LNCS 2763, pp. 236-243, 2003.

[3] Yu. Serdyuk "MC# 2.0: a language for concurrent distributed programming based on .NET". .NET Technologies 2006, 4th International Conference, Plzen, Czech Republic (http://www.mcsharp.net/publications/NET2006.pdf).

[4] A. D. Gordon, P. D. Hankin "A Concurrent Object Calculus: Reduction and Typing". Electr. Notes Theor. Comput. Sci. 16(3): (1998).

[5] A. Ravara, V. T. Vasconcelos."Typing Non-uniform Concurrent Objects". CONCUR 2000. LNCS 1877, pp. 474-488, 2000.

[6] C. Fournet, C. Laneve, L. Maranget, D. Remy. "Inheritance in the Join Calculus". J. Logic and Algeb.Prog. 57(2003)23-69.

[7] C. Fournet, C. Laneve, L. Maranget, D. Remy. "Implicit typing à la ML for the join-calculus". CONCUR '97, LNCS 1243, pp. 196-212, 1997.

[8] V. Saraswat, R. Jagadeesan. "Concurrent Clustered Programming". CONCUR 2005. LNCS 3653, pp. 353-367, 2005.