プログラミング－言語・基礎・実践－　1－4
（1991. 4. 26）

並行オブジェクト計算のための小さな形式系

本田　耕平　　所　真理雄
慶応義塾大学　理工学部　計算機学科

われわれは，並行オブジェクト指向計算のエッセンスを理論的な文脈のなかで明確にするために，ロビン・ミルナーのπ計算を基盤とした小さな形式系を構成し，さらにその計算概念を反映した興味深い意味論枠組を提示する．非同期的通信を表現した操作的意味論とそれに基盤をおく模倣的前順序は，従来の意味論的枠組に比べて極めて大きな飛躍を意味するだけでなく，プロセス計算という形式系の持つ可能性に対するわれの視点を大きく変化させ，並行オブジェクトのダイナミズムを体現した新たな形式系の構成への優れたヒントとなることが期待される．

A Small Calculus for Concurrent Objects (revised version)*

Kohei Honda and Mario Tokoro†
Department of Computer Science, Keio University,
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

We study a fragment of Milner's *pi*-calculus with the intention of understanding the notion of concurrent object-based computing in the theoretical setting. We construct a formal system from the original one as an elementary expression of the notion of "objects" and "asynchronous messages", not by adding but by reducing it to a smaller system while retaining the computational power almost fully. What is interesting is that our asynchronous bisimulation is broader than the traditional bisimulation and moreover congruence relation.

A Small Calculus for Concurrent Objects (revised version)*

Kohei Honda and Mario Tokoro†
Department of Computer Science, Keio University,
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

We study a fragment of Milner's *pi*-calculus with the intention of understanding the notion of concurrent object-based computing in the theoretical setting. We construct a formal system from the original one as an elementary expression of the notion of "objects" and "asynchronous messages", not by adding but by reducing it to a smaller system while retaining the computational power almost fully. What is interesting is that our asynchronous bisimulation is broader than the traditional bisimulation and moreover congruence relation.

1 Introduction

This paper introduces a calculus of concurrent objects, which tries to represent the essence of concurrent object-based computing in a simplest possible formal system. The concurrent object-oriented computing is enchanting software modeling methodology based on the abstraction of "objects" and "messages" [17, 15]. At the same time, we do not have the foundational formalism for concurrent object-based computing, which has been harmful in various theoretical and pragmatic investigations. This paper presents an attempt to construct the core theory of concurrent object-based computing, which is a simple formalism of *objects* and *communication* just as Lambda-calculus is that of *functions* and *application*. As such, our hope is that the formalism will provide the basis for further clarification and development of concurrent object-based computing.

Two important precursors should be noticed before we present our formalism. One is the study of the actor model by Carl Hewitt and his colleagues [5, 4, 1], which contributed to the conceptual framework of our formalism. Another is Milner's π-calculus, proposed in [10] and reformulated in [11], which is a basis of theoretical construction of our formalism. For details of relationship between our calculus and Milner's calculus, please consult [8].

The structure of the rest of the paper is as follows. Section 2 presents the conceptual framework of our formalism called *migration-based computing*, which underlies our

*The previous version of this paper appeared in Proceedings of the Workshop for Object-based Concurrent Systems, Ottawa, 1990.

†Also with Sony Computer Science Laboratory Inc. 3-14-13 Higashi-Gotanda, Shinagawa-ku, Tokyo, 141, Japan

theoretical development. Section 3 defines the syntax and operational semantics for our formal system. Section 4 discusses several remaining topics. Finally Section 5 concludes the paper.

2 The Computational Framework

This section presents the unusual computational framework called *migration-based computing*[13, 6, 14], which underlies our theoretical construction described later. It is unique in that it positions the inherent concurrent, distributed computational space, where distance and locality are dual important notions essential in any aspects of computation, at the center of the *general* study of concurrent computation, not only of the study of physical distributed computation space. There we no longer consider that computational space consists of processing nodes and communication networks. We instead have *the field.*

Let us envisage that a computational space is a horizontal, continuous extension called *the computational field.* The field is supposed to give computational power and migration capability to entities functioning on it. We suppose that each point in the field is given its own address (which we suppose globally distinct). On this field, there are two kinds of entities, *objects* and *messages*, both being supposed to be very fine-grained and primitive. An object has a *name* and *child entities.* The name should be the same as some address (please note that, in general, there can be multiple objects with the same name). The object should be situated in some neighborhood of this address, and does not voluntarily move around. In contrast, messages are more active entities. A message has the name of its target object, and a value to transmit to the target. Once generated, it will *migrate* to its target address. Then it will actively *search* for the target object around the address, to *interact* with it. Now computation proceeds as follows;

1. When interaction takes place, the message is assimilated into the object and the value is transmitted. Possibly using this value, the object generates child entities into the computational field. The original message and object just disappear.

2. If objects will be generated, they are situated in that local field, waiting for messages, and if messages are generated, each will migrate to its respective target.

The name of one object may be inherited, in some cases atavistically, by its descendants, which means a rather loose kind of *persistence.* As these descendants will be situated in the same local space, the number of objects in the neighborhood may grow as time proceeds. Thus some kind of biological populations (or *colonies*) are generated here and there. We can identify such a colony as a kind of *local configuration* in the global computation space.

The most important point in this framework is its departure from the foregoing *node-network* metaphor of computation and communication. This results in a clean way of representing asynchronous messages as *syntactic terms* in our formalism presented later. Another point is its emphasis on *locality* and *communication*, which primarily distinguishes our framework from Chemical Abstract Machine [2]. For more detailed discussions, see [8]. Also see [16] for application of the "field" concept to load balancing strategy in distributed computation environments.

3 The Formal System

Our basic framework is that very small entities, some being objects and some being messages, interact with one another in a horizontal extension called the computational field. Apart from the formalization of such subtle mechanisms as migration etc., the framework can be very cleanly represented as a formal system of computation. This is what we will show in the subsequent exposition, a formal system[1] born from the marriage between clean expression of dynamic concurrent computation in Milner's π-calculus and our migration framework of computation.

3.1 Syntax and Structural Congruence

In the syntax definition below, we assume an infinite set of *labels*, written in non-capital letters or sometimes non-capital strings.

Definition 1 *Syntax by extended BNF grammar.*

O	::=	$\leftarrow a:v$	(a migrating message)	(1)
	|	$a:\{x.\ O\}$	(an object)	(2)
	|	$O\ ,\ O$	(entities running concurrently)	(3)
	|	$P(v)(a,b,c,....)$	(instantiate an object definition)	(4)
	|	$\|y\|\ O$	(private labels)	(5)
	|	Nil	(a null term)	(6) ■

We will call the expression (a *term*) of the form (1) and (2), *a message term* and *an object term*, meaning a message and an object, respectively. We call "a" in (2) a *handle*, meaning a communication handle. Subterms of O in (2) are called *descendants* of $a:\{x.\ O\}$. The meaning of each term above is clear, except (4) and (5). First, in (4), we are assuming existence of a *defining expression* for object behaviour, in the form of

$$P(a)(y_1,y_2,\ldots) \stackrel{\text{def}}{=} a:\{x.\ O\} \qquad \text{(where } y_1,y_2, \text{ etc. are distinct.)}$$

Here we suppose that each of free labels in the right hand side should be one of $y_1,y_2,\ldots$ Second, in (5), private labels are *binding* to the labels which are not yet bound in the subsequent expression. That is, "$|a|$" in "$|a|\ E$" will bind the appearances of "a" which are free in expression "E". The private labels are used instead of **let** statements (of the form "let x = new ...") which are found in various actor-based languages as in π-calculus [10]. They function as new label creation in reality. It is called *p-binding*. In addition, "x" in "$a:\{x.\ O\}$" binds the free occurrences of x in O. This binding is called *v-binding*. For these bindings, we assume standard substitution rules of variables.

The below is the syntactic equivalence rule called structural congruence. It was first proposed and successfully exploited by Milner in [11], and we follow his formulation closely here.

[1]The previous version of this formal system is called ACF, but the current version remains anonymous for the time being.

Definition 2 The structural congruence *is the smallest congruence relation over terms which includes:*

$$
\begin{array}{rcll}
O_1 & \equiv & O_2 & \text{(if } O_1 \text{ is an alpha conversion of } O_2) \quad (1) \\
O_1,\, O_2 & \equiv & O_2,\, O_1 & (2\text{-a}) \\
O_1,\, (O_2,\, O_3) & \equiv & (O_1,\, O_2),\, O_3 & (2\text{-b}) \\
P(v)(a_1,..) & \equiv & A\{v/x\}\{a_1,../y_1,..\} & \text{(when } P(x)(y_1,..) \stackrel{\text{def}}{=} A) \quad (3) \\
|x||y|O & \equiv & |y||x|O & (4) \\
|x|(O_1,\, O_2) & \equiv & O_1,\, |x|O_2 & \text{(if x is not free in } O_1) \quad (5) \\
O, Nil & \equiv & O & (6)
\end{array}
$$

■

Among all, (5) is the most important, in that it states the meaning of p-binding. Informally speaking, this rule says that p-binding means that the label is globally new in the whole space of the field of computation.

3.2 Semantics

A configuration (or a program) is a collection of terms. Its meaning can be defined in many ways, but a possibly most general steps to define the operational semantics for our system may be as follows;

1. Define the basic interaction transition rules, specifying internal interaction sequences.
2. Based on the transition rules, construct its behaviour rules which stipulate the possible reduction sequences when the configuration is composed with any possible configurations, labeled according to the category of such transitions.

The below is the bahavioural transition rules constructed following the framework above (see [7] for details). We will use $a:v$, $a:|v|$, $a(x)$, *and* τ as labels for transitions.

Definition 3 *Behaviour transition of a term is a labeled transition relation, which can be inferred by the following rules. It embodies the notion of asynchronous message passing as foundation of our calculus.*

$$
\begin{array}{lll}
\text{IN}: & \dfrac{-}{P \xrightarrow{a(v)} P,\leftarrow a:v} & \\
\text{OUT}: & \dfrac{-}{\leftarrow a:v \xrightarrow{a:v} \text{Nil}} & \\
\text{COM}: & \dfrac{-}{\leftarrow a:v, a:\{x.\, O\} \xrightarrow{\tau} O\{v/x\}} & \text{(=internal computation)} \\
\text{RES}: & \dfrac{O \xrightarrow{l} O'}{|x|O \xrightarrow{l} |x|O'} & \text{(if } x \notin \mathit{labels}(l)) \\
\text{OPEN}: & \dfrac{O \xrightarrow{a:v} O'}{|v|O \xrightarrow{a:|v|} |v|O'} & \text{(if } a \neq v) \\
\text{PAR(1)}: & \dfrac{O_1 \xrightarrow{l} O_1'}{O_1,\, O_2 \xrightarrow{l} O_1',\, O_2} & \text{(if } l \text{ not of the form } x:|v|) \\
\text{PAR(2)}: & \dfrac{O_1 \xrightarrow{x:|v|} O_1'}{O_1,\, O_2 \xrightarrow{x:|v|} O_1',\, O_2} & \text{(if } v \text{ not free in } O_2) \\
\text{STRUCT}: & \dfrac{O_1' \equiv O_1,\ O_1 \xrightarrow{l} O_2,\ O_2 \equiv O_2'}{O_1' \xrightarrow{l} O_2'} & \blacksquare
\end{array}
$$

The transition relation denotes action sequences the one configuration can have when it is composed with some configuration. In terms of migration-based computing, it denotes sequences of consumption and generation of messages inside the configuration when it is situated at some local space in the field (where many other configurations exist here and there). Especially please note that, in **IN** rule, we boldly introduce *asynchronous message reception*, which means that a configuration can just get messages from outside, without any interaction happening. Not only does this express our notion of asynchronous message passing loyally, but also this formulation of operational semantics is found to be more robust and general than synchronous one, especially in that the abstraction level is *higher* in its expression of causal connection and hence concurrency. See [8] for details.

Based on this behaviour transition, we can define the standard strong and weak bisimulation. Firstly, the "strong" bisimulation requires that two configurations have exactly the same set of transition sequences.

Definition 4 *A term O_1 is said to be strongly bisimilar to O_2 (written $O_1 \sim O_2$) iff: whenever $O_1 \xrightarrow{l} O_1'$ then for some $O_2' \sim O_1'$, $O_2 \xrightarrow{l} O_2'$, and the same with O_1 and O_2 reversed.* ■

The second one is more important, as here lies the core of Milner's approach in the semantics of concurrent systems, hiding unobservable internal computation in the semantic schemes.

Definition 5 *A term O_1 is said to be weakly bisimilar to O_2 (written $O_1 \approx O_2$) iff: whenever $O_1 \xrightarrow{l} O_1'$ then for some $O_2' \approx O_1'$, $O_2 \stackrel{\hat{l}}{\Longrightarrow} O_2'$, and the same with O_1 and O_2 reversed.* ■

The relation $\stackrel{l}{\Longrightarrow}$ stands for $\xrightarrow{\tau}{}^* \xrightarrow{l} \xrightarrow{\tau}{}^*$, and $\hat{l}$ means the null if $l = \tau$ and l itself otherwise. The definition says that two programs are equivalent if and only if an outside "observer" cannot distinguish them by any interaction sequences. In terms of our migration framework, this bisimulation only pays attention to interaction between the local configuration and the outside environment in the form of message exchanges. This weak bisimulation is strictly more general than its "synchronous" counterpart [8].

Strong and weak bisimulation are defined to be the largest such relations, whose existence we can prove. More interesting "relativized" versions of these equivalences are discussed in e.g. [12, 9].

3.3 Object-Orientation and Naming

The readers may wonder how the usual notion of "objects" can be compared to the terse expression of "object terms" in our calculus. Indeed the careful examination of the behaviour transition rules will soon lead us to the observation that the behaviour rules *do not prevent multiple object terms from having the same object id*, which may seem severe violation of the usual principle of object-orientation. Also it is obvious that our object terms may or may not be *persistent*. That is, regeneration of an object (i.e. generation of a descendant object with the same handle but possibly with the different behaviour), may be or may not be carried out. Please note how these "object-oriented" notion is closely related with the framework of *naming of the descendant object terms* in our calculus.

While the condition of identity preservation and strict persistence can be stipulated easily as naming principles, our present formal system is based on a much looser framework regarding these points. It is called *Local Naming Constraint* and is given as follows.

Definition 6 *A program P conforms to* the local naming constraint, *written as $LN(P)$, iff none of handles of object terms are v-bound in P.* ■

What is the significance of this naming framework? There are two important consequences from this definition:

- If free handles of object terms in one configuration do not overlap with those in another, they will not overlap in any future transitions.
- It does not prevent multiple objects from sharing the same handles.

Thus multiple objects with the same name can coexist, but only within some local boundary of the computational space. Hence the collection of behaviors of objects with the same name are *locally* given, rather than from the arbitrary configurations. Preliminary investigation utilizing this concept will be presented in [8], where we show the relationship between the naming concept and controllability notion. We contend that, while the local naming constraint is much looser than the foregoing framework of concurrent objects, it does offer us another authentic notion of "objects", which is based on *locality* rather than strict identity. See also [3] for related discussions from the pragmatic concerns. Further consequences of this "object-orientation" in our calculus is, however, to be clarified in our future investigations.

4 Remaining Topics

We did not talk much about the expressive power of our formal language, which deserves close scrutiny just because it is virtually a rather small subset of π-calculus. In reality, we can encode essential primitive data/control structures using only basic constructs of our calculus. In other words, our formal system is computationally complete, and moreover can express "concurrent objects" very naturally in its terse syntax (see [8]). Another notable fact is all the constructs of the full π-calculus except the *matching* operator and *unguarded summation* can be loyally mapped to our calculus (up to weak bisimulation). In a sense, our formal system is "purely asynchronous" version of π-calculus, without no embedded synchronization constructs unless absolutely necessary (i.e. only in the case of value passing and subsequent bindings). Yet it is equipped with the almost similar expressive power. The relationship with π-calculus and our "subset" calculus, however, needs further careful investigations.

There are other interesting topics concerning our formal system, such as: investigation of various typing schemes; construction of denotational framework; development of formal specification-verification methodologies; the logical architecture for concurrent objects computing based on our formal theory; and comparisons with other framework of concurrent computation, especially those using streams for communication such as A'UM [18]. Some of these issues are being actively investigated in our laboratory, and their results will be discussed in our subsequent papers in near future.

5 Conclusion

In this paper we presented a simple and rigorous formalism for concurrent objects computing based on the formalism of Milner's π-calculus. Coupled with its underlying framework of migration-based computing, it opens new possibilities for formal, conceptual, and pragmatic aspects of concurrent object-based computing. The formal system is powerful enough to represent dynamic computation scheme of concurrent object-based computing. Being a significantly simple formalism, we hope that it can serve as one of basic strata for investigation of theoretical and other aspects of concurrent object-based computing.

Concluding this paper, we would like to express many thanks to Carl Hewitt for stimulating discussions during his stay in Keio University from 1989 to 1990, to researchers who gave us beneficial suggestions in the course of development of our theory, to Vasco Vasconcelos for good advices and productive criticisms on the paper, and to all the members of our laboratory for kind assistance and cheers.

References

[1] Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Berry, G. and Boudol, G., *The Chemical Abstract Machine.* Proc 17 the Annual Symposium on Principles of Programming Languages, 1990.

[3] Chien, A., *Concurrent Aggregates: Using Multiple-Access Data Abstractions to Manage Complexity in Concurrent Programs.* to appear in Proceedings of OBCS Workshop 1990, SIGPLAN NOTICES.

[4] Clinger, W. *Foundations of Actor Semantics.* AI-TR-633, MIT Artificial Intelligence Laboratory.

[5] Hewitt, C., *Viewing Control Structures as Patterns of Passing Messages.* Artificial Intelligence, 1977.

[6] Honda, K., Tokoro, M., *A Report On Language PROTO and Its Underlying Computation Model.* KEIO-CS-1989-1, April 1989. With annotations in October 1990 as KEIO-CS-1990-1.

[7] Honda, K., *Defining Operational Semantics from Reduction Relation.* a manuscript, November 1990

[8] Honda, K., Tokoro, M., *Objects and Calculi.* to appear in Proc. of ECOOP 91, LNCS ??.

[9] Larsen, K.G., *Context-Dependent Bisimulation Between Processes.* PhD thesis, University of Edinburgh, 1986.

[10] Milner, R., Parrow, J.G. and Walker, D.J., *A Calculus of Mobile Processes. Part I and II.* ECS-LFCS-89-85/86, Edinburgh University, 1989

[11] Milner, R., *Functions as Processes.* Rapports de Recherche No.1154, INRIA-Sophia Antipolis, February 1990.

[12] Nierstrasz, O., *Towards a Type Theory for Active Objects.* in [15].

[13] Tokoro, M., *Issues In Object-Oriented Distributed Computing.* in: *Proceedings of 4th Conference of Japan Society for Software Science and Technology*, September 1988. (in Japanese, English version available)

[14] Tokoro, M., *Computational Field Model: Toward a New Computing Model/Methodology for Open Distributed Environment.* The 2nd IEEE Workshop on Future Trends in Distributed Computing Systems, Cairo, 1990.

[15] Tsichritzis, D., ed. *Object Management.* Centre Universitaire D'informatique, Universite de Geneve, July 1990.

[16] Uehara,M. and Tokoro,M, *An Adaptive Load Balancing Method in the Computational Field Model.* in this volume.

[17] Yonezawa, A., and Tokoro, M., *Object-Oriented Concurrent Programming.* MIT Press, 1986.

[18] Yoshida, K., *A'UM: A Stream-Based Concurrent Object-Oriented Programming Language.* Ph.d.Thesis, Keio University, 1990.