STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXII**, Number 2, 2017
DOI: 10.24193/subbi.2017.2.08

METRIC DENOTATIONAL SEMANTICS FOR REMOTE PROCESS DESTRUCTION AND CLONING

ENEIA NICOLAE TODORAN

Abstract. We present a denotational semantics designed with continuations for a concurrent language providing a mechanism for synchronous communication, together with constructions for process creation, remote process destruction and cloning. We accomplish the semantic investigation in the mathematical framework of complete metric spaces.

1. Introduction

We study the semantics of a concurrent language $\mathcal{L}_{syn}^{pc}$ providing a mechanism for synchronous communication, together with constructions for process creation, remote process destruction and cloning. We design a denotational semantics for $\mathcal{L}_{syn}^{pc}$ by using the *continuation semantics for concurrency (CSC)* technique [16]. Following [4], we accomplish the semantic investigation in the mathematical framework of complete metric spaces.

The central characteristic of the CSC technique is the modeling of continuations as application-specific structures of computations, where by computation we understand a partially evaluated denotation (meaning function). The CSC technique was introduced and developed in a series of works [16, 7, 8]. A comparison between CSC and the classic direct approach to concurrency semantics [4] is provided in [16, 7].

To illustrate how synchronous interactions can be modeled with CSC, in Section 3 we start with a language $\mathcal{L}_{syn}$ that is very simple but provides a synchronization mechanism between concurrent components. In Section 3 we provide a denotational semantics designed with CSC for $\mathcal{L}_{syn}$. $\mathcal{L}_{syn}$ is a *uniform* language in the sense that its elementary statements are uninterpreted

Received by the editors: November 30, 2017.

2010 *Mathematics Subject Classification.* 68Q55, 68Q85, 68N15, 68M14.

1998 *CR Categories and Descriptors.* D.3.2 [**Software**]: Programming Languages – *Concurrent, distributed, and parallel languages*; F.3.2 [**Theory of Computation**]: Logics and Meanings of Programs – *Denotational semantics*.

Key words and phrases. metric semantics, continuation semantics for concurrency, remote process destruction, remote process cloning.

symbols taken from a given alphabet. $\mathcal{L}^{pc}_{syn}$ is a *non-uniform* language: in general, in $\mathcal{L}^{pc}_{syn}$ the behavior of an elementary statement depends upon the current state of a program. The terminology *uniform* vs *non-uniform* language is also used, e.g., in [4, 18].

The language $\mathcal{L}^{pc}_{syn}$ is studied in Section 4. $\mathcal{L}^{pc}_{syn}$ provides CSP-like synchronous communication [10]. $\mathcal{L}^{pc}_{syn}$ also provides constructions for process creation, process destruction and process cloning. In $\mathcal{L}^{pc}_{syn}$ a process can not only commit suicide or clone itself, but it can also kill or clone any other process in the system. Process creation is a well known control concept encountered both at operation system level and in concurrent programming. Process destruction and process cloning are operations that can be encountered at operating system level, in some coordination languages [11], or in distributed object oriented and multi agent systems such as Obliq [5] and IBM Java Aglets [12, 19]. The former operation kills a parallel running process and is similar to the "`kill -9`" system call in Unix. The latter operation creates an identical copy of a (parallel) running process.

For the development of our ideas we have chosen the mathematical framework of metric semantics [4], where the main mathematical tool is Banach's fixed point theorem. We need the theory developed in [2] for solving reflexive domain equations as continuations in the CSC approach are elements of a complete space which is the solution of a domain equation where the domain variable occurs in the left-hand side of a function space construction.

1.1. **Contribution.** We present a denotational (mathematical) semantics for a concurrent language $\mathcal{L}^{pc}_{syn}$ incorporating advanced control mechanisms for remote process creation, destruction and cloning. The denotational semantics is designed with metric spaces [4] and continuation semantics for concurrency (CSC) [16], a technique providing sufficient flexibility for handling the advanced control concepts incorporated in $\mathcal{L}^{pc}_{syn}$. Various semantic models for languages with process creation are presented, e.g., in [1, 3, 4, 15]. However, as far as we know, this is the first paper presenting a denotational (mathematical) semantics for remote process destruction and cloning.

2. Preliminaries

The notation $(x \in)X$ introduces the set X with typical element x ranging over X. For X a set we denote by $\mathcal{P}_\pi(X)$ the collection of all subsets of X which have property π. For example, $\mathcal{P}_{finite}(X)$ is the set of all finite subsets of X. If $f : X \to X$ and $f(x) = x$ we call x a *fixed point* of f. When this fixed point is unique (see 2.1) we write $x = fix(f)$. The notions of *partial order* and *total* or *simple order* are assumed to be known. We recall that, given a partially ordered set $(X, \leq_X)$, an element $x \in X$ is said to be *maximal* if there

are no elements strictly greater than x in X, that is if $x \leq_X y$ then $y \leq_X x$ in which case $x = y$.

Let $(x \in)X, (y \in)Y, (z \in)Z$, $f \in X \to Y$ and $g \in X \to Y \to Z$. The functions $(f \mid x \mapsto y) : X \to Y$ and $(g \mid x, y \mapsto z) : X \to Y \to Z$ are defined as follows:

$$(f \mid x \mapsto y)(x') = \begin{cases} y & \text{if} \quad x' = x \\ f(x') & \text{if} \quad x' \neq x \end{cases}$$

$$(g \mid x, y \mapsto z) = (g \mid x \mapsto (g(x) \mid y \mapsto z))$$

$(f \mid x \mapsto y)$ is a variant of the function f which behaves like f almost everywhere, except for point x where $(f \mid x \mapsto y)$ yields y. Instead of $((f \mid x_1 \mapsto y_1) \cdots \mid x_n \mapsto y_n)$ we write $(f \mid x_1 \mapsto y_1 \mid \cdots \mid x_n \mapsto y_n)$.

Following [4] the study presented in this paper takes place in the mathematical framework of 1-bounded complete metric spaces. We assume known the notions of metric and ultrametric space, isometry (distance preserving bijection between metric spaces; we denote it by '$\cong$') and completeness of metric spaces. If (X, d_X), (Y, d_Y) are metric spaces we recall that a function $f : X \to Y$ is a *contraction* if $\exists c \in \mathbf{R}$, $0 \leq c < 1$: $\forall x_1, x_2 \in X$: $d_Y(f(x_1), f(x_2)) \leq c \cdot d_X(x_1, x_2)$. Also, f is called *non-expansive* if $d_Y(f(x_1), f(x_2)) \leq d_X(x_1, x_2)$. We denote the set of all c-contracting (nonexpansive) functions from X to Y by $X \xrightarrow{c} Y$ ($X \xrightarrow{1} Y$).

Theorem 2.1. *(Banach) Let (X, d_X) be a non-empty complete metric space. Each contracting function $f : X \to X$ has a* unique *fixed point.*

For any set $(a, b \in)A$ the so-called *discrete metric* d_A is defined as follows: $d_A(a, b) =$ if $a = b$ then 0 else 1. (A, d_A) is a complete ultrametric space.

Definition 2.1. *Let $(X, d_X), (Y, d_Y)$ be (ultra) metric spaces. On $(x \in)X$, $(f \in)X \to Y$ (the function space), $((x, y) \in)X \times Y$ (the cartesian product), $(u, v \in)X \sqcup Y$ (the disjoint union) and on $(U, V \in)\mathcal{P}(X)$ (the power set of X) we can define the following metrics:*

(a) $d_{\frac{1}{2}\cdot X} : X \times X \to [0, 1]$, $\quad d_{\frac{1}{2}\cdot X}(x_1, x_2) = \frac{1}{2} \cdot d_X(x_1, x_2)$

(b) $d_{X \to Y} : (X \to Y) \times (X \to Y) \to [0, 1]$
$d_{X \to Y}(f_1, f_2) = sup_{x \in X} d_Y(f_1(x), f_2(x))$

(c) $d_{X \times Y} : (X \times Y) \times (X \times Y) \to [0, 1]$
$d_{X \times Y}((x_1, y_1), (x_2, y_2)) = max\{d_X(x_1, x_2), d_Y(y_1, y_2)\}$

(d) $d_{X \sqcup Y} : (X \sqcup Y) \times (X \sqcup Y) \to [0, 1]$*:*
$d_{X \sqcup Y}(u, v) =$
if $u, v \in X$ then $d_X(u, v)$ else if $u, v \in Y$ then $d_Y(u, v)$ else 1

(e) $d_H : \mathcal{P}(X) \times \mathcal{P}(X) \to [0, 1]$ *is the Hausdorff distance defined by:*

$$d_H(U,V) = max\{sup_{u\in U} d(u,V), sup_{v\in V} d(v,U)\}$$
$$where\ d(u,W) = inf_{w\in W} d(u,w)\ (by\ convention\ sup\emptyset = 0,\ inf\emptyset = 1).$$

Given a metric space (X, d_X) a subset A of X is called *compact* whenever each sequence in A has a convergent subsequence with limit in A. We will use the abbreviations $\mathcal{P}_{co}(\cdot)$ ($\mathcal{P}_{nco}(\cdot)$) to denote the power set of compact (non-empty and compact) subsets of '$\cdot$'.

Remark 2.1. *Let $(X, d_X), (Y, d_Y), d_{\frac{1}{2}\cdot X}, d_{X\to Y}, d_{X\times Y}$, $d_{X\sqcup Y}$ and d_H be as in definition 2.1. In case d_X, d_Y are ultrametrics, so are $d_{\frac{1}{2}\cdot X}, d_{X\to Y}$, $d_{X\times Y}$, $d_{X\sqcup Y}$ and d_H. If in addition $(X, d_X), (Y, d_Y)$ are complete then $(X, d_{\frac{1}{2}\cdot X})$, $(X \to Y, d_{X\to Y})$, $(X \xrightarrow{1} Y, d_{X\to Y})$, $(X\times Y, d_{X\times Y})$, $(X\sqcup Y, d_{X\sqcup Y})$, $(\mathcal{P}_{co}(X), d_H)$ and $(\mathcal{P}_{nco}(X), d_H)$ are also complete metric spaces. In the sequel we will often suppress the metrics part in domain definitions. In particular we will write $\frac{1}{2}\cdot X$ instead of $(X, d_{\frac{1}{2}\cdot X})$.*

2.1. **Structure of Continuations.** In the CSC approach a continuation is an application-specific structure of computations. Intuitively, a CSC-based model is a semantic formalization of a process scheduler which repeatedly selects and activates computations contained in a continuation [16]. Let $(x \in)\mathbf{X}$ be a complete metric space. Following [16, 7, 8] we define the domain of CSC with the aid of a set $(\alpha \in)Id$ of *(process) identifiers* and we use the following notation:

$$\{\!|\mathbf{X}|\!\} \stackrel{not.}{=} \mathcal{P}_{finite}(Id) \times (Id \to \mathbf{X})$$

We let π range over $\mathcal{P}_{finite}(Id)$. Let $(\pi, \phi) \in \{\!|\mathbf{X}|\!\}$, where ϕ ranges over $Id \to \mathbf{X}$. We define $id : \{\!|\mathbf{X}|\!\} \to \mathcal{P}_{finite}(Id)$, $id(\pi, \phi) = \pi$. We use the following abbreviations: $(\pi, \phi)(\alpha) \stackrel{not.}{=} \phi(\alpha)$, $(\pi, \phi) \setminus \pi' \stackrel{not.}{=} (\pi \setminus \pi', \phi)$, $((\pi, \phi) \mid \alpha \mapsto x) \stackrel{not.}{=} (\pi \cup \{\alpha\}, (\phi \mid \alpha \mapsto x))$. The operations $id, (\cdot)(\alpha)$, $(\cdot) \setminus \pi$ and $(\cdot \mid \alpha \mapsto x)$ are further explained in [7, 8].

We treat (π, ϕ) as a 'function' with finite graph $\{(\alpha, \phi(\alpha)) \mid \alpha \in \pi\}$, thus ignoring the behaviour of ϕ for any $\alpha \notin \pi$ (π is the 'domain' of the 'function'). Essentially, a structure (π, ϕ) is a finite partially ordered *bag* (or multiset)[1] of computations.

We also use the following notation:

$$[\mathbf{X}] \stackrel{not.}{=} \mathbf{N} \times (\mathbf{N}^+ \to \mathbf{X})$$

We let ι range over $(\iota \in)\mathbf{N}$ and φ range over $(\varphi \in)\mathbf{N} \to \mathbf{X}$. $\mathbf{N}$ is the set of natural numbers, and $\mathbf{N}^+ = \mathbf{N} \setminus \{0\}$ (the set of positive natural numbers).

[1] A *partially ordered multiset* is a more refined structure; see, e.g., chapter 16 of [4].

We use a structure $(\iota, \varphi) \in [\mathbf{X}]$ to model a *stack* of elements of the type $\mathbf{X}$. We define $[]_{(\cdot)} : [\mathbf{X}]$, $\mathsf{empty}(\cdot) : [\mathbf{X}] \to \mathit{Bool}$, $(\cdot : \cdot) : \mathbf{X} \times [\mathbf{X}] \to [\mathbf{X}]$, $\mathsf{hd}(\cdot) : [\mathbf{X}] \to (\mathbf{X} \cup \{\uparrow\})$ and $\mathsf{tl}(\cdot) : [\mathbf{X}] \to ([\mathbf{X}] \cup \{\uparrow\})$ as follows

$$[]_x = (0, \lambda\iota.x)$$
$$\mathsf{empty}(\iota, \varphi) = (\iota = 0)$$
$$x : (\iota, \varphi) = (\iota + 1, (\varphi \mid \iota + 1 \mapsto x))$$

$$\mathsf{hd}(\iota, \varphi) = \begin{cases} \uparrow & \mathsf{if} \quad \iota = 0 \\ \varphi(\iota) & \mathsf{if} \quad \iota > 0 \end{cases}$$

$$\mathsf{tl}(\iota, \varphi) = \begin{cases} \uparrow & \mathsf{if} \quad \iota = 0 \\ (\iota - 1, \varphi) & \mathsf{if} \quad \iota > 0 \end{cases}$$

Remarks 2.1.

(a) *If we endow* Id, $\mathcal{P}_{finite}(Id)$ *and* $\mathbf{N}$ *with discrete ultrametrics, then* $\{\!|\mathbf{X}|\!\}$ *and* $[\mathbf{X}]$ *are also a complete ultrametric space.* $\{\!|\mathbf{X}|\!\}$ *and* $[\mathbf{X}]$ *are composed spaces built up using the composite metrics of definition 2.1.*

(b) *We use the set* $(\alpha \in)Id$ *(of process identifiers) together with a function* $\nu : \mathcal{P}_{finite}(Id) \to Id$*, defined such that* $\nu(A) \notin A$*, for every* $A \in \mathcal{P}_{finite}(Id)$*. A possible example of such a set* Id *and function* ν *is* $Id = \mathbf{N}$ *and* $\nu(A) = max(A) + 1$*, with* $\nu(\emptyset) = 0$*.*

(c) *Throughout this paper the symbol* $\uparrow$ *denotes an* undefined value*.*

(d) $[]_x$ *is an* empty stack*, for any* $x \in \mathbf{X}$*.*

3. A simple uniform language with synchronization

In this section we apply the CSC technique in the definition of a denotational semantics for a simple concurrent language $\mathcal{L}_{syn}$ with *synchronization*. $\mathcal{L}_{syn}$ is essentially based on Milner's CCS [14]. The language $\mathcal{L}_{syn}$ provides atomic actions, recursion, action prefixing (in the form $a;s$), nondeterministic choice ($s_1 + s_2$) and parallel composition ($s_1 \| s_2$). We assume given two sets $(c \in)Sync$ and $(\overline{c} \in)\overline{Sync} = \{\overline{c} \mid c \in Sync\}$ of *synchronization actions*, and a set $(b \in)IAct$ of *internal actions*. We define $(a \in)Act = IAct \cup Sync \cup \overline{Sync}$, and let τ be a special symbol, $\tau \notin Act$. We also assume given a set $(x \in)PVar$ of procedure variables. Synchronization in $\mathcal{L}_{syn}$ is achieved by the execution of a pair $\overline{c}, c$. First, the $\overline{c}$-step is executed. It is followed immediately by the corresponding c-step. *There are no actions interspersed between* $\overline{c}$ *and* c*.* The order is important here: the $\overline{c}$-step is always executed first. A $\overline{c}$-step is an abstract model of a send operation, while a c-step is an abstract model of a

receive operation. The approach to recursion in $\mathcal{L}_{syn}$ is based on declarations and guarded statements [4].

Definition 3.1. *(Syntax of $\mathcal{L}_{syn}$)*

(a) *(Statements)* $s(\in Stat) ::= a \mid a;s \mid x \mid s+s \mid s\|s$
(b) *(Guarded statements)* $g(\in GStat) ::= a \mid a;s \mid g+g \mid g\|g$
(c) *(Declarations)* $(D \in)Decl = PVar \to GStat$; *following* [4] *we assume a fixed declaration* D*!*
(d) *(Programs)* $(\rho \in)Prog = Decl \times Stat$

The denotational semantics function $\mathcal{D}$ is of the type $(\mathcal{D} \in)Sem_D = Stat \to \mathbf{D}$, where $\mathbf{D}$ is defined by the following system of domain equation (isometry between complete metric spaces):

$$\mathbf{D} \cong (Id \times \mathbf{Kont}) \xrightarrow{1} \Gamma \to \mathbf{P}$$
$$(\gamma \in)\Gamma = \{\uparrow_\Gamma\} \cup \overline{Sync}$$
$$(\kappa \in)\mathbf{Kont} = \{\!|\frac{1}{2} \cdot \mathbf{D}|\!\}$$
$$(p \in)\mathbf{P} = \mathcal{P}_{nco}((IAct \cup \{\tau\})^\infty)$$

The construction $\{\!|\frac{1}{2} \cdot \mathbf{D}|\!\}$ was explained in Section 2.1. In the 'equations' above, the sets Id, $\{\uparrow_\Gamma\} \cup \overline{Sync}$ and $IAct \cup \{\tau\}$ are endowed with the discrete metric (which is an ultrametric). The elements γ of the set $(\gamma \in)(\{\uparrow_\Gamma\} \cup \overline{Sync})$ contain *synchronization information.*

The space $(IAct \cup \{\tau\})^\infty$ contains all finite (possibly empty) and infinite sequences over $(IAct \cup \{\tau\})$. $(IAct \cup \{\tau\})^\infty$ is an instance of the following:

Definition 3.2. *Let* $(x \in)\mathbf{X}$ *be a nonempty complete space. The space* $\mathbf{X}^\infty$ *is defined by the equation* $\mathbf{X}^\infty \cong \{\epsilon\} \sqcup (\mathbf{X} \times \frac{1}{2} \cdot \mathbf{X}^\infty)$. ϵ *models the* empty sequence. *The elements of* $\mathbf{X}^\infty$ *are finite or infinite sequences over* $\mathbf{X}$. *Instead of* $(x_1, (x_2, \ldots, (x_n, \epsilon)\ldots))$, *and* $(x_1, (x_2, \ldots))$ *we write* $x_1x_2\ldots x_n$, *and* $x_1x_2\ldots$, *respectively. We use the symbol "$\cdot$" as a concatenation operator over sequences. In particular we write* $x \cdot q = (x, q)$, *for any* $x \in \mathbf{X}$ *and* $q \in \mathbf{X}^\infty$*; we also write* $x \cdot p = \{x \cdot q \mid q \in p\}$ *for any* $x \in \mathbf{X}$ *and* $p \in \mathcal{P}_{nco}(\mathbf{X}^\infty)$.

Definition 3.3. *We let* q *range over* $\mathbf{Q} = (IAct \cup \{\tau\})^\infty$. *We define* $+, \oplus : \mathbf{P} \times \mathbf{P} \to \mathbf{P}$ *as follows:*

$$p_1 + p_2 = \{q \mid q \in p_1 \cup p_2, q \neq \tau\} \cup \{\tau \mid \tau \in (p_1 \cap p_2)\}$$
$$p_1 \oplus p_2 = \{q \mid q \in p_1 \cup p_2, q \neq \epsilon\} \cup \{\epsilon \mid \epsilon \in (p_1 \cap p_2)\}$$

For any $\gamma \in (\{\uparrow_\Gamma\} \cup \overline{Sync})$ *we also define* $\oplus^\gamma : \mathbf{P} \times \mathbf{P} \to \mathbf{P}$ *by:*

$$p_1 \oplus^\gamma p_2 = \text{ if } \gamma = \uparrow_\Gamma \text{ then } p_1 + p_2 \text{ else } p_1 \oplus p_2$$

Definition 3.4. *(Denotational semantics for $\mathcal{L}_{syn}$) Let $C_+ : \mathbf{Kont} \to \mathbf{P}$ and $C_\oplus : \mathbf{Kont} \to (\overline{Sync} \times Id) \to \mathbf{P}$ be given by:*

$$C_+(\kappa) = \mathsf{if}\ (id(\kappa) = \emptyset)\ \mathsf{then}\ \{\epsilon\}\ \mathsf{else}\ +_{\alpha \in id(\kappa)}\ \kappa(\alpha)(\alpha, \kappa \setminus \{\alpha\})(\uparrow_\Gamma)$$
$$C_\oplus(\kappa)(\overline{c}, \overline{\alpha}) = \mathsf{if}\ (id(\kappa) \setminus \{\overline{\alpha}\} = \emptyset)\ \mathsf{then}\ \{\epsilon\}$$
$$\mathsf{else}\ \oplus_{\alpha \in (id(\kappa) \setminus \{\overline{\alpha}\})}\ \kappa(\alpha)(\alpha, \kappa \setminus \{\alpha\}, \overline{c})$$

We define the denotational semantics function $\mathcal{D} : Stat \to \mathbf{D}$ as follows:

$$\begin{array}{rcl}
\mathcal{D}(b)(\alpha,\kappa)(\gamma) & = & \mathsf{if}\ \gamma = \uparrow_\Gamma\ \mathsf{then}\ \tau{\cdot}b{\cdot}C_+(\kappa)\ \mathsf{else}\ \{\epsilon\} \\
\mathcal{D}(b;s)(\alpha,\kappa)(\gamma) & = & \mathsf{if}\ \gamma = \uparrow_\Gamma\ \mathsf{then}\ \tau{\cdot}b{\cdot}C_+(\kappa \mid \alpha \mapsto \mathcal{D}(s))\ \mathsf{else}\ \{\epsilon\} \\
\mathcal{D}(\overline{c})(\alpha,\kappa)(\gamma) & = & \mathsf{if}\ \gamma = \uparrow_\Gamma\ \mathsf{then}\ \tau{\cdot}C_\oplus(\kappa)(\overline{c}, \alpha)\ \mathsf{else}\ \{\epsilon\} \\
\mathcal{D}(\overline{c}; s)(\alpha,\kappa)(\gamma) & = & \mathsf{if}\ \gamma = \uparrow_\Gamma\ \mathsf{then}\ \tau{\cdot}C_\oplus(\kappa \mid \alpha \mapsto \mathcal{D}(s))(\overline{c}, \alpha)\ \mathsf{else}\ \{\epsilon\} \\
\mathcal{D}(c)(\alpha,\kappa)(\gamma) & = & \mathsf{if}\ \gamma = \uparrow_\Gamma\ \mathsf{then}\ \{\tau\} \\
 & & \mathsf{else\ if}\ \gamma = \overline{c}\ \mathsf{then}\ \tau{\cdot}C_+(\kappa)\ \mathsf{else}\ \{\epsilon\} \\
\mathcal{D}(c; s)(\alpha,\kappa)(\gamma) & = & \mathsf{if}\ \gamma = \uparrow_\Gamma\ \mathsf{then}\ \{\tau\} \\
 & & \mathsf{else\ if}\ \gamma = \overline{c}\ \mathsf{then}\ \tau{\cdot}C_+(\kappa \mid \alpha \mapsto \mathcal{D}(s))\ \mathsf{else}\ \{\epsilon\} \\
\mathcal{D}(x)(\alpha,\kappa)(\gamma) & = & \mathcal{D}(D(x))(\alpha,\kappa)(\gamma) \\
\mathcal{D}(s_1 + s_2)(\alpha,\kappa)(\gamma) & = & \mathcal{D}(s_1)(\alpha,\kappa)(\gamma) \oplus^\gamma \mathcal{D}(s_2)(\alpha,\kappa)(\gamma) \\
\mathcal{D}(s_1 \parallel s_2)(\alpha,\kappa)(\gamma) & = & \mathcal{D}(s_1)(\alpha_1, (\kappa \mid \alpha_2 \mapsto \mathcal{D}(s_2)))(\gamma) \oplus^\gamma \\
 & & \mathcal{D}(s_2)(\alpha_2, (\kappa \mid \alpha_1 \mapsto \mathcal{D}(s_1)))(\gamma)
\end{array}$$

where in the last clause $\alpha_1 = \nu(id(\kappa)), \alpha_2 = \nu(id(\kappa) \cup \{\alpha_1\})$.

Let $b_0 \in IAct$ be a distinguished internal action. Let $\kappa_0 = (\emptyset, \lambda\alpha.\mathcal{D}(b_0))$ and $\alpha_0 = \nu(\emptyset)$. We define $\mathcal{D}[\![\cdot]\!] : Stat \to \mathbf{P}$ by

$$\mathcal{D}[\![s]\!] = \mathcal{D}(s)(\alpha_0, \kappa_0)(\uparrow_\Gamma)$$

Following [16], we use the term *process* to denote a computation (partially evaluated denotation) contained in a continuation. Synchronization is modeled as in [16]. Some explanations may help.

- In the yield of $\mathcal{D}$ successful synchronization is modeled by two consecutive τ-steps ($\tau\tau$), which correspond to some pair $\overline{c}, c$ of synchronization actions. Single τ steps are used to model *deadlock*. They can only be produced by unsuccessful synchronization attempts and they are removed from the yield of $\mathcal{D}$ as long as there are alternative computations. This is expressed in the definition of the operator $+$. The operator $\oplus$ describes the behavior of the system in those states where a synchronization attempt occurred. Thus a τ-step has been produced and its pair is expected. No other action is possible. If some process produces a τ step then the computation continues. The computation stops only if all processes are unable to produce the expected τ-step. This is marked by the empty sequence ϵ in the yield of $\mathcal{D}$. In those states where synchronization succeeds by the

contribution of some concurrent process, $\oplus$ removes the eventual ϵ's from the final yield of $\mathcal{D}$. It is easy to check that the operators $+, \oplus$ and $\oplus^\gamma$ (for any $\gamma \in (\{\uparrow_\Gamma\} \cup \overline{Sync})$) are well-defined, nonexpansive, associative, commutative and idempotent [4].

- Note that the execution of an internal action $b \in IAct$ is also preceded by a τ-step. In the CSC approach this is necessary only if we want to obtain a denotational model which is *correct* with respect to a corresponding operational model; further explanations are provided in [16].
- A process can not synchronize with itself. The function $C_\oplus$ receives as parameter the process identifier of the current process, and chooses some other process for synchronization.

Remark 3.1. *$\mathcal{D}$ can be formally defined as fixed point of an appropriate higher order contraction. In Section 4 we give the details of such a proof for a more complex language. For $\mathcal{L}_{syn}$, the proof can proceed by induction on the following complexity measure: $c : Stat \to \mathbf{N}$, $c(a) = c(a;s) = 1, c(x) = 1 + c(D(x))$, $c(s_1 + s_2) = c(s_1 \| s_2) = 1 + max\{c(s_1), c(s_2)\}$; the mapping c is well-defined due to our restriction to guarded recursion* [4].

Examples 3.1.

- $\mathcal{D}[\![\overline{c} \parallel c]\!] = \mathcal{D}(\overline{c} \parallel c)(\alpha_0, \kappa_0)(\uparrow_\Gamma)$
  $= \mathcal{D}(\overline{c})(\alpha_1, (\kappa_0 \mid \alpha_2 \mapsto \mathcal{D}(c)))(\uparrow_\Gamma) \oplus^{\uparrow_\Gamma} \mathcal{D}(c)(\alpha_2, (\kappa_0 \mid \alpha_1 \mapsto \mathcal{D}(\overline{c})))(\uparrow_\Gamma)$
  $= \tau \cdot C_\oplus(\kappa_0 \mid \alpha_2 \mapsto \mathcal{D}(c))(\overline{c}, \alpha_1) + \{\tau\}$
  $= \tau \cdot \mathcal{D}(c)(\alpha_2, \kappa_0')(\overline{c}) + \{\tau\} = \tau \cdot \tau \cdot C_+(\kappa_0') + \{\tau\}$ *[$id(\kappa_0') = \emptyset$]*
  $= \{\tau\tau\} + \{\tau\} = \{\tau\tau\}$
  *where $\alpha_0 \in Id$ and $\kappa_0 \in$ **Kont** are as in Definition 3.4, $\alpha_1 = \nu(id(\kappa_0))$, $\alpha_2 = \nu(id(\kappa_0) \cup \{\alpha_1\})$, and $\kappa_0' = (\kappa_0 \mid \alpha_2 \mapsto \mathcal{D}(c)) \setminus \{\alpha_2\}$.*
- $\mathcal{D}[\![b_1 + b_2]\!] = \{\tau b_1, \tau b_2\}$
- $\mathcal{D}[\![(\overline{c} + b) \parallel c]\!] = \{\tau\tau, \tau b\tau\}$

4. Remote Process Destruction and Cloning

The CSC technique can be used to design denotational (compositional) semantics for various advanced control concepts, including: synchronous and asynchronous communication [16], multiparty interactions [17, 8], maximal parallelism [6, 9] and systems with dynamic hierarchical structure [9]. In this work we use CSC to design a denotational semantics for an imperative concurrent language $\mathcal{L}_{syn}^{pc}$ providing synchronous CSP-like synchronous communication [10] together with constructions for process creation, and remote process destruction and cloning.

We assume given a class of *variables* $(v \in) Var$, a set $(e \in) Exp$ of *expressions*, and a set $(x \in) PVar$ of *procedure variables*. We also assume given a class

$(c \in) Chan$ of *communication channels*. We assume that the evaluation of an expression ($e \in Exp$) always terminates and delivers a value in some set $(\alpha \in) Val$. The set Val of values is assumed to be countably infinite.

Remark 4.1. *The constructs for process control (*new, kill, clone*) operate with* process identifiers*, which are elements of the given countably infinite set $(\alpha \in) Id$ introduced in Section 2.1. For simplicity (and without loss of generality), in the rest of this section* we assume that the class $(\alpha \in) Id$ of process identifiers coincides with the class Val of vales: $Id = Val$.[2]

Definition 4.1. *We define the syntax of $\mathcal{L}^{pc}_{syn}$ by the following components:*

(a) *(Statements)* $s(\in Stat) ::= a \mid x \mid s + s \mid s;s$
(b) *(Guarded statements)* $g(\in GStat) ::= a \mid g + g \mid g;s$
(c) *(Declarations)* $(D \in) Decl = PVar \to GStat$
(d) *(Programs)* $(\rho \in) Prog = Decl \times Stat$

where $a(\in AStat)$ is given by:

$$a ::= \mathsf{skip} \mid v := e \mid c!e \mid c?v \mid v := \mathsf{new}(s) \mid \mathsf{kill}(e) \mid v := \mathsf{clone}(e)$$

We assume an approach to recursion based on declarations and guarded statements (as in the previous section). Without loss of generality [4], in the rest of this section *we assume a fixed declaration $D \in Decl$ and in all contexts we refer to this fixed D*. In $\mathcal{L}^{pc}_{syn}$ we have assignment ($v := e$), recursion, sequential composition ($s; s$), nondeterministic choice ($s + s$), CSP-like synchronous communication (given by the statements $c!e$ and $c?v$) and constructions for process creation ($v :=$ new(s)), remote process destruction (kill(e)) and remote process cloning ($v{:=}$ clone(e)). The net effect of a construct $v :=$ new(s) is to create a new process with body s that runs in parallel with all other processes in the system. A new *process identifier* is automatically generated and assigned to v. In a kill(e) or $v{:=}$ clone(e) statement, the expression e is evaluated to some value α, which is interpreted as a process identifier.[3] The execution of a statement kill(e) kills the parallel runing process with identifier α. When a $v{:=}$ clone(e) statement is executed, a new process - identical to the one with identifier α - is created and its identifier is assigned to v. The constructs $c!e$ and $c?v$ are as in Occam [13]. Synchronized execution of two actions $c!e$ and $c?v$ occurring in two parallel processes, results in the transmission of the current value of e along the channel c from the process executing

[2]For example, we could put $Id = Val = \mathbf{N}$ ($\mathbf{N}$ is the set of natural numbers) in which case Exp would be a class of numeric expressions. However, it is straightforward to extend the semantic model by using different support sets for the class of values and the class of process identifiers.

[3]The statements kill(e) and $v{:=}$ clone(e) are inoperative if the value of the expression e (in the current state) is not a valid process identifier.

the $c!e$ (send) statement to the process executing the $c?v$ (receive) statement. The latter assigns the received value to the variable v.

4.1. **Denotational Semantics.** In the definition of the denotational semantics for $\mathcal{L}_{syn}^{pc}$ we use the set $(\alpha \in)Id$ of process identifiers and the constructions $\{\!|\cdot|\!\}$ and $[\cdot]$ introduced in Section 2.1. In $\mathcal{L}_{syn}^{pc}$ each process has its own local data. Values can be communicated between processes but there is no shared memory area. We define a class $(\sigma \in)State = Id \to Var \to Val$ of (distributed) *states*. The meaning of (the local) variables of a process with identifier α is given by $\sigma(\alpha)$. The evaluation of expressions in $\mathcal{L}_{syn}^{pc}$ is modeled by a given valuation $V : Exp \to (Var \to Val) \to Val$. We recall that we take $Val = Id$.

We design a denotational semantics function $\mathcal{D}$ for $\mathcal{L}_{syn}^{pc}$. The type of $\mathcal{D}$ is $\mathcal{D} : Sem_D = Stat \to \mathbf{D}$, where:

$$(\psi \in)\mathbf{D} \cong (Id \times \mathbf{Kont}) \xrightarrow{1} (\Omega \times State) \to \mathbf{P}_D$$

$$(\kappa \in)\mathbf{Kont} = \{\!|\ [\frac{1}{2} \cdot \mathbf{D}]\ |\!\}$$

$$(\omega \in)\Omega = \{\uparrow_\Omega\} \cup (Chan \times Val)$$

$$(q \in)\mathbf{Q} = (\{\tau\} \cup State)^\infty$$

$$(p \in)\mathbf{P}_D = \mathcal{P}_{nco}(\mathbf{Q})$$

The construction $(\{\tau\} \cup State)^\infty$ was introduced in Definition 3.2. We assume that $\tau \notin State$. For easier readability, we denote typical elements (c, α) of $Chan \times Val(\subseteq \Omega)$ by $c!\alpha$.

Remark 4.2. *In the definition of the domain of continuatins* **Kont** *we use the constructions* $\{\!|\cdot|\!\}$ *and* $[\cdot]$ *introduced in Section 2.1. A continuation of type* $\mathbf{Kont} = \{\!|\ [\frac{1}{2} \cdot \mathbf{D}]\ |\!\}$ *is essentially a bag (multiset) of stacks of computations. An element of the type* $[\frac{1}{2} \cdot \mathbf{D}]$ *is a stack of computations (denotations). In this section we use the term* process *when referring to an element of the type* $[\frac{1}{2} \cdot \mathbf{D}]$*. A stack of computations of the type* $[\frac{1}{2} \cdot \mathbf{D}]$ *represents a process (an execution thread with a local state) executed in parallel with all the other processes contained in a continuation.*

The domain of computations (denotations) $\mathbf{D}$ is given by a recursive domain equation. In the domain equations given above, the sets Id, Ω, and $State$ (and $\{\tau\} \cup State$) are endowed with discrete metrics (which are ultrametrics).

According to [2] the solutions for **D** and **Kont** are obtained as complete ultrametric spaces.

The denotational semantics function is defined below as the fixed point of an appropriate higher-order mapping. In Definition 4.2 the operators presented in Definition 3.3 are adapted to $\mathcal{L}^{pc}_{syn}$.

Definition 4.2. **4.2.1 Definition** $+, \oplus : \mathbf{P}_D \times \mathbf{P}_D \to \mathbf{P}_D$ *are defined by:*

$$p_1 + p_2 = \{q \mid q \in p_1 \cup p_2, q \neq \tau\} \cup \{\tau \mid \tau \in (p_1 \cap p_2)\}$$
$$p_1 \oplus p_2 = \{q \mid q \in p_1 \cup p_2, q \neq \epsilon\} \cup \{\epsilon \mid \epsilon \in (p_1 \cap p_2)\}$$

For any $\omega \in \Omega$ *we also define* $\oplus^\omega : \mathbf{P} \times \mathbf{P} \to \mathbf{P}$ *by:*

$$p_1 \oplus^\omega p_2 = \mathsf{if}\ \omega = \uparrow_\Omega\ \mathsf{then}\ p_1 + p_2\ \mathsf{else}\ p_1 \oplus p_2$$

The operators $+$, $\oplus$ and $\oplus^\omega$ are well-defined, nonexpansive, associative, commutative and idempotent [4].

Definition 4.3. *(Denotational semantics* $\mathcal{D}$ *for* $\mathcal{L}^{pc}_{syn}$*) We define* $C_+ : \mathbf{Kont} \to State \to \mathbf{P}_D$ *and* $C_\oplus : \mathbf{Kont} \to (\Omega \times Id \times State) \to \mathbf{P}_D$ *as follows:*

$$\begin{aligned}
&C_+(\kappa)(\sigma) = \\
&\quad \mathsf{let}\ \overline{\kappa} = \kappa \setminus \{\alpha' \mid \mathsf{empty}(\kappa(\alpha'))\}\ \mathsf{in} \\
&\quad \mathsf{if}\ id(\overline{\kappa}) = \emptyset\ \mathsf{then}\ \{\epsilon\} \\
&\quad \mathsf{else}\ +_{\alpha \in id(\overline{\kappa})}\ \mathsf{hd}(\overline{\kappa}(\alpha))\,(\alpha, (\overline{\kappa} \mid \alpha \mapsto \mathsf{tl}(\overline{\kappa}(\alpha)))(\uparrow_\Omega, \sigma) \\
&C_\oplus(\kappa)(\omega, \overline{\alpha}, \sigma) = \\
&\quad \mathsf{let}\ \overline{\kappa} = \kappa \setminus \{\alpha' \mid \mathsf{empty}(\kappa(\alpha'))\}\ \mathsf{in} \\
&\quad \mathsf{if}\ id(\overline{\kappa}) \setminus \{\overline{\alpha}\} = \emptyset\ \mathsf{then}\ \{\epsilon\} \\
&\quad \mathsf{else}\ \oplus_{\alpha \in (id(\overline{\kappa}) \setminus \{\overline{\alpha}\})}\ \mathsf{hd}(\overline{\kappa}(\alpha))\,(\alpha, (\overline{\kappa} \mid \alpha \mapsto \mathsf{tl}(\overline{\kappa}(\alpha)))(\omega, \sigma)
\end{aligned}$$

Let $u \in UStat$ *be given by:*

$$u ::= \mathsf{skip} \mid v := e \mid c!e \mid v := \mathsf{new}(s) \mid \mathsf{kill}(e) \mid v := \mathsf{clone}(e)$$

The statements of the subclass $UStat \subseteq AStat$ *can not be executed in those states where a communication attempt occurred.*

Let α^* *be some distinguished value* $(\alpha^* \in) Val$*. Let* ψ^* *be some distinguished computation* $\psi^* \in \mathbf{D}$*.*

We define $\Psi \in Sem_D \to Sem_D$ *for* $S \in Sem_D (= Stat \to \mathbf{D})$ *by:*

$$
\begin{array}{rcl}
\Psi(S)(u)(\alpha,\kappa)(c!\alpha,\sigma) & = & \{\epsilon\} \quad \text{for any } u \in UStat \\
\Psi(S)(\mathsf{skip})(\alpha,\kappa)(\uparrow_\Omega,\sigma) & = & \tau{\cdot}\sigma{\cdot}C_+(\kappa)(\sigma) \\
\Psi(S)(v := e)(\alpha,\kappa)(\uparrow_\Omega,\sigma) & = & \tau{\cdot}\sigma_{assign}{\cdot}C_+(\kappa)(\sigma_{assign}) \\
\Psi(S)(c!e)(\alpha,\kappa)(\uparrow_\Omega,\sigma) & = & \tau{\cdot}C_\oplus(\kappa)(c!V(e)(\sigma(\alpha)),\alpha,\sigma) \\
\Psi(S)(c?v)(\alpha,\kappa)(\omega,\sigma) & = & \left\{\begin{array}{ll} \{\tau\} & \text{if} \quad \omega = \uparrow_\Omega \\ \sigma_{rcv}{\cdot}C_+(\kappa)(\sigma_{rcv}) & \text{if} \quad \omega = c!\alpha' \\ \{\epsilon\} & \text{if} \quad \omega = c'!\alpha' \\ & \quad\ c' \neq c \end{array}\right. \\
\Psi(S)(v := \mathsf{new}(s))(\alpha,\kappa)(\uparrow_\Omega,\sigma) & = & \tau{\cdot}\sigma_{new}{\cdot}C_+(\kappa \mid \alpha_\nu \mapsto S(s) : []_{\psi^*})(\sigma_{new}) \\
 & & \text{where } \alpha_\nu = \nu(id(\kappa)) \\
\Psi(S)(\mathsf{kill}(e))(\alpha,\kappa)(\uparrow_\Omega,\sigma) & = & \tau{\cdot}\sigma{\cdot}C_+(\kappa \setminus \{\overline{\alpha}\})(\sigma) \\
 & & \text{where } \overline{\alpha} = V(e)(\sigma(\alpha)) \\
\Psi(S)(v := \mathsf{clone}(e))(\alpha,\kappa)(\uparrow_\Omega,\sigma) & = & \tau{\cdot}\sigma_{clone}{\cdot}C_+(\kappa \mid \alpha_\nu \mapsto \kappa(\overline{\alpha}))(\sigma_{clone}) \\
 & & \text{where } \alpha_\nu = \nu(id(\kappa)) \\
 & & \qquad\ \overline{\alpha} = V(e)(\sigma(\alpha)) \\
\Psi(S)(x)(\alpha,\kappa)(\omega,\sigma) & = & \Psi(S)(D(x))(\alpha,\kappa)(\omega,\sigma) \\
\Psi(S)(s_1 + s_2)(\alpha,\kappa)(\omega,\sigma) & = & \Psi(S)(s_1)(\alpha,\kappa)(\omega,\sigma) \ \oplus^\omega \\
 & & \Psi(S)(s_2)(\alpha,\kappa)(\omega,\sigma) \\
\Psi(S)(s_1; s_2)(\alpha,\kappa)(\omega,\sigma) & = & \Psi(S)(s_1)(\alpha,(\kappa \mid \alpha \mapsto S(s_2) : \kappa(\alpha)))(\omega,\sigma)
\end{array}
$$

where $\sigma_{assign} = (\sigma \mid \alpha, v \mapsto V(e)(\sigma(\alpha)))$ *in the semantic equation for* $v := e$, $\sigma_{rcv} = (\sigma \mid \alpha, v \mapsto \alpha')$ *in the (second) semantic equation for* $c?v$, $\sigma_{new} = ((\sigma \mid \alpha, v \mapsto \alpha_\nu) \mid \alpha_\nu \mapsto \lambda v'.\alpha^*)$ *in the semantic equation for* $v := \mathsf{new}(s)$, *and* $\sigma_{clone} = ((\sigma \mid \alpha, v \mapsto \alpha_\nu) \mid \alpha_\nu \mapsto \sigma(\overline{\alpha}))$ *in the equation for* $v := \mathsf{clone}(e)$.

We recall that ψ^* *is a distinguished computation* $\psi^* \in \mathbf{D}$. *The notation* $[]_{\psi^*}$ *was introduced in Section 2.1 as a mean to construct an empty stack.*

We put $\mathcal{D} = fix(\Psi)$. *Let* $\alpha_0 = \nu(\emptyset)$ *and* $\kappa_0 = (\{\alpha_0\}, \lambda\alpha.[]_{\psi^*})$. *We define* $\mathcal{D}[\![\cdot]\!] : Stat \to State \to \mathbf{P}_D$ *by*

$$\mathcal{D}[\![s]\!](\sigma) = \mathcal{D}(s)(\alpha_0,\kappa_0)(\uparrow_\Omega,\sigma)$$

The technique used to model synchronous interactions was introduced in [16] and was already illustrated in this paper in Section 3. In the semantic equation for process creation $v := \mathsf{new}(s)$ a new process executing the computation $\mathcal{D}(s)$ is started in parallel with all processes contained in the continuation. Process destruction is handled in the sematic equation for $\mathsf{kill}(e)$ by removing the process with identifier $\overline{\alpha} = V(e)(\sigma(\alpha))$, where α is the identifier of the process which executes the statement $\mathsf{kill}(e)$ in the current state σ. Process cloning is handled in the sematic equation for $v := \mathsf{clone}(e)$ by starting a clone of the process with identifier $\overline{\alpha} = V(e)(\sigma(\alpha))$, where α is the identifier of the

process which executes the statement $v := \mathsf{clone}(e)$ and σ is the current state of the distributed system.

The denotational semantics $\mathcal{D}$ is defined as the (unique) fixed point of the higher-order mapping Ψ. Definition 4.3 is justified by Lemma 4.1, Lemma 4.2 and Banach's fixed point theorem 2.1. Similar lemmas are given in [16]. We omit the proof of 4.1. To illustrate a proof technique specific of metric semantics we present the proof of Lemma 4.2(c) by using and inductive argument. For inductive reasonings, in the case of the language $\mathcal{L}^{pc}_{syn}$ one can use the following complexity measure $c_s : Stat \to \mathbf{N}$ $c_s(a) = 1$, for any $a \in AStat$, $c_s(x) = 1 + c_s(D(x))$, $c_s(s_1; s_2) = 1 + c_s(s_1)$ and $c_s(s_1 + s_2) = 1 + max\{c_s(s_1), c_s(s_2)\}$. The mapping c_s is well defined due to our restriction to guarded recursion [4].

Lemma 4.1. *The mappings C_+ and $C_\oplus$ (as introduced in Definition 4.3) are well-defined. Also, for any $\kappa_1, \kappa_2 \in \mathbf{Kont}$ we have:*

(a) $d(C_+(\kappa_1,)C_+(\kappa_2)) \leq 2{\cdot}d(\kappa_1, \kappa_2)$ *and*
(b) $d(C_\oplus(\kappa_1), C_\oplus(\kappa_2)) \leq 2{\cdot}d(\kappa_1, \kappa_2)$.

Lemma 4.2. *For any $S{\in}Sem_D, s{\in}Stat, \alpha{\in}Id, \kappa{\in}\mathbf{Kont}, \omega{\in}\Omega, \sigma{\in}State$:*

(a) $\Psi(S)(s)(\alpha, \kappa)(\omega, \sigma) \in \mathbf{P}_D$ *(it is well-defined),*
(b) $\Psi(S)(s)$ *is nonexpansive (in κ) and*
(c) Ψ *is $\frac{1}{2}$-contractive in S.*

Proof We only prove Lemma 4.2(c). It suffices to show that

$$d(\Psi(S_1)(s)(\alpha, \kappa)(\omega, \sigma), \Psi(S_2)(s)(\alpha, \kappa)(\omega, \sigma)) \leq \tfrac{1}{2} \cdot d(S_1, S_2).$$

We proceed by induction on $c_s(s)$. Two subcases.

Case $s = (v := \mathsf{clone}(e))$
$d(\Psi(S_1)(v := \mathsf{clone}(e))(\alpha, \kappa)(\omega, \sigma),$
$\quad \Psi(S_2)(v := \mathsf{clone}(e))(\alpha, \kappa)(\omega, \sigma))$
$= 0 \leq \frac{1}{2} \cdot d(S_1, S_2)$

Case $s = x$
$d(\Psi(S_1)(x))(\alpha, \kappa)(\omega, \sigma), \Psi(S_2)(x)(\alpha, \kappa)(\omega, \sigma))$
$= d(\Psi(S_1)(D(x)))(\alpha, \kappa)(\omega, \sigma),$
$\qquad \Psi(S_2)(D(x))(\alpha, \kappa)(\omega, \sigma))$
$\qquad\quad [c_s(D(x)) < c_s(x)$, ind. hypothesis]
$\leq \frac{1}{2} \cdot d(S_1, S_2)$

□

Remark 4.3. *When the CSC technique is employed in the semantic design, (groups of) computations contained in a continuations can be manipulated as data. By expoiting this facility in this paper we offer a denotational (compositional) semantics for remote process destruction and cloning. Our attempts*

to model such remote control operations by using only classic compositional techniques have failed. In the classic direct approach to concurrency [4] *the semantic designer defines the various operators for parallel composition as functions that manipulate final semantic values belonging to some power domain construction. The meaning of a process appears in the final yield of a denotational mapping interleaved with the meanings of other parallel processes. The problem with this approach is that the remote control operations considered in this paper may be executed long after the creation of the process. It it does not seem possible to extract somehow the meaning of a process from an element of a power domain, in order to remove (kill) it or to clone it. On the contrary, in the CSC approach the semantic designer operates with partially evaluated meaning functions (contained in continuations) which can easily be manipulated to model such remote control operations.*

Example 4.1. *In this example we assume that Exp is a class of numeric expressions and put* $Val = \mathbf{N}$. *Consider the* $\mathcal{L}^{pc}_{syn}$ *program statement* $s(\in Stat)$

$$s = v_{new} := \mathsf{new}(c!1); ((v_{clone} := \mathsf{clone}(v_{new}); \mathsf{kill}(v_{new}); c?v) + v := 2)$$

Let $\sigma \in \Sigma$, $\alpha^{new}_{\nu} = \nu(id(\kappa_0)) = \nu(\{\alpha_0\})$ *and* $\alpha^{clone}_{\nu} = \nu(\{\alpha_0, \alpha^{new}_{\nu}\})$, *where* $\kappa_0 \in \mathbf{Kont}$ *and* $\alpha_0 \in Id$ *are as in Definition 4.3. Let* $\overline{\sigma}_{new} = ((\sigma \mid \alpha_0, v_{new} \mapsto \alpha^{new}_{\nu}) \mid \alpha^{new}_{\nu} \mapsto \lambda v'.\alpha^*)$, $\overline{\sigma}_{clone} = ((\overline{\sigma}_{new} \mid \alpha_0, v_{clone} \mapsto \alpha^{clone}_{\nu}) \mid \alpha^{clone}_{\nu} \mapsto \overline{\sigma}_{new}(\alpha^{new}_{\nu}))$, $\overline{\sigma}_{kill} = \overline{\sigma}_{clone}$, $\overline{\sigma}_{rcv} = (\overline{\sigma}_{kill} \mid \alpha_0, v \mapsto 1)$, $\overline{\sigma}_{assign} = (\overline{\sigma}_{new} \mid \alpha_0, v \mapsto 2)$. *It is easy to check that:*

$$\mathcal{D}[\![s]\!](\sigma) = \mathcal{D}(s)(\alpha_0, \kappa_0)(\uparrow_\Omega, \sigma) = \tau \cdot \overline{\sigma}_{new} \cdot \mathcal{D}(s_1)(\alpha_0, \kappa_1)(\uparrow_\Omega, \overline{\sigma}_{new})$$

where $s_1 = (v_{clone} := \mathsf{clone}(v_{new}); \mathsf{kill}(v_{new}); c?v) + v := 2$, *and* $\kappa_1 = (\kappa_0 \mid \alpha_0 \mapsto []_{\psi^*} \mid \alpha^{new}_{\nu} \mapsto \mathcal{D}(c!1) : []_{\psi^*})$. *We have:*

$$\begin{aligned}
&\mathcal{D}(s_1)(\alpha_0, \kappa_1)(\uparrow_\Omega, \overline{\sigma}_{new}) \\
&= \mathcal{D}(v_{clone} := \mathsf{clone}(v_{new}); \mathsf{kill}(v_{new}); c?v)(\alpha_0, \kappa_1)(\uparrow_\Omega, \overline{\sigma}_{new}) \oplus^{\uparrow_\Omega} \\
&\quad \mathcal{D}(v := 2)(\alpha_0, \kappa_1)(\uparrow_\Omega, \overline{\sigma}_{new}) \\
&= \mathcal{D}(v_{clone} := \mathsf{clone}(v_{new}))(\alpha_0, \kappa_2)(\uparrow_\Omega, \overline{\sigma}_{new}) + \{\tau \overline{\sigma}_{assign} \tau\}
\end{aligned}$$

where $\kappa_2 = (\kappa_0 \mid \alpha_0 \mapsto \mathcal{D}(\mathsf{kill}(v_{new}); c?v) : []_{\psi^*} \mid \alpha^{new}_{\nu} \mapsto \mathcal{D}(c!1) : []_{\psi^*})$. *Note that a deadlock is detected after the execution of the assignment statement* $v := 2$ *because (when* $v := 2$ *is selected for execution) the computation* $\mathcal{D}(c!1)$ *contained in the continuation has no synchronization counterpart .*

The execution of the statement $v_{clone} := \mathsf{clone}(v_{new})$ *creates an copy of the process with identifier* α^{new}_{ν} *which will run in parallel with the other processes.*

$$\mathcal{D}(v_{clone} := \mathsf{clone}(v_{new}))(\alpha_0, \kappa_2)(\uparrow_\Omega, \overline{\sigma}_{new}) = \tau \cdot \overline{\sigma}_{clone} \cdot C_+(\kappa_3)(\overline{\sigma}_{clone})$$

where $\kappa_3 = (\kappa_0 \mid \alpha_0 \mapsto \mathcal{D}(\mathsf{kill}(v_{new}); c?v) : []_{\psi^*} \mid \alpha^{new}_{\nu} \mapsto \mathcal{D}(c!1) : []_{\psi^*} \mid \alpha^{clone}_{\nu} \mapsto \mathcal{D}(c!1) : []_{\psi^*})$.

After the cloning operation, the execution of the statement $\mathsf{kill}(v_{new})$ *removes the process with identifier* α_ν^{new} *from the continuation. Next, after a synchronization step the computation terminates. In the end we obtain:*

$$\mathcal{D}[\![s]\!](\sigma) = \{\tau\overline{\sigma}_{new}\tau\overline{\sigma}_{clone}\tau\overline{\sigma}_{kill}\tau\overline{\sigma}_{rcv}, \tau\overline{\sigma}_{new}\tau\overline{\sigma}_{assign}\tau\}$$

where $s = v_{new} := \mathsf{new}(c!1); ((v_{clone} := \mathsf{clone}(v_{new}); \mathsf{kill}(v_{new}); c?v) + v := 2)$.

5. Conclusion

The CSC technique can be used to design denotational (compositional) semantics for various advanced control concepts, including: synchronous and asynchronous communication [16, 7], maximal parallelism [6, 9], and multiparty interactions [17, 8]. In this work we present a denotational semantics designed with metric spaces and continuations for a concurrent language providing constructions for CSP-like synchronous communication in combination with constructions for process creation, and remote process control (remote process destruction and cloning). Various denotational models for process creation have been developed [1, 15, 3, 4]. However, we are not aware of any paper reporting a denotational model for remote process destruction and process cloning.

The use of continuation semantics for concurrency (CSC) technique [16, 7] proved to be fruitful. In the CSC approach the semantics of remote process destruction and cloning can easily be modeled by appropriate manipulations of the computations contained in continuations. We think that the CSC technique could be used to model remote process control operations in combination with various interaction mechanisms, including remote procedure call and ADA-like rendezvous. The rendezvous programming concept was studied by using techniques from metric semantics in several papers [15, 3, 4]. In the near future we intend to study the formal relationship between the denotational and the operational semantics of remote process destruction and cloning also by using techniques from metric semantics [4].

References

[1] P. America and J.W. de Bakker, Designing Equivalent Semantic Models for Process Creation, *Theoretical Computer Science*, vol. 60, 1988, pp. 109–176.

[2] P. America and J.J.M.M. Rutten, Solving Reflexive Domain Equations in a Category of Complete Metric Spaces, *Journal of Computer and System Sciences*, vol. 39, 1989, pp. 343–375.

[3] J.W. de Bakker and E.P. de Vink, Rendez-vous with Metric Semantics, *New Generation Computing*, vol. 12, 1993, pp. 53–90.

[4] J.W. de Bakker and E.P. de Vink, *Control Flow Semantics,* MIT Press, 1996.

[5] L. Cardelli, A Language with Distributed Scope, *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pp. 286–297, ACM Press, 1995.
[6] G Ciobanu and EN Todoran, Relating Two Metric Semantics for Parallel Rewriting of Multisets, *Proceedings of 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2012)*, pp. 273–280, IEEE Computer Press, 2012.
[7] G. Ciobanu and E.N. Todoran, Continuation Semantics for Asynchronous Concurrency, *Fundamenta Informaticae*, vol. 131(3–4), 2014, pp. 373–388.
[8] G Ciobanu and EN Todoran, Continuation Semantics for Concurrency with Multiple Channels Communication, *Formal Methods and Software Engineering - Proceedings of 17th International Conference on Formal Engineering Methods (ICFEM 2015)*, *Lecture Notes in Computer Science*, vol. 9407, 2015, pp. 400–416.
[9] G Ciobanu and EN Todoran, Continuation Semantics for Dynamic Hierarchical Systems, *Proceedings of 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2015)*, pp. 281–288, IEEE Computer Press, 2015.
[10] C.A.R. Hoare, *Communicating Sequential Processes,* Prentice Hall, 1985.
[11] A.A. Holzbacher, A Software Environment for Concurrent Coordinated Programming, *Proc. 1st Int. Conference on Coordination Languages and Systems*, *Lecture Notes in Computer Science*, vol. 1061, 1996, pp. 249–267.
[12] D.B. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets,* Addison Wesley, 1998.
[13] INMOS Ltd, *Occam Programming Manual,* Prentice-Hall, 1984.
[14] R. Milner, *Communication and Concurrency,* Prentice-Hall, 1989.
[15] J.J.M.M. Rutten, Semantic Correctnes for a Parallel Object-Oriented Language, *SIAM Journal of Computing*, vol. 19, 1990, pp. 341–383.
[16] E.N. Todoran, Metric Semantics for Synchronous and Asynchronous Communication: a Continuation-based Approach, *Electronic Notes in Theoretical Computer Science*, vol. 28, 2000, pp. 101–127.
[17] E.N. Todoran and N. Papaspyrou, Experiments with Continuation Semantics for DNA Computing, *Proceedings of the IEEE 9th International Conference on Intelligent Computer Communication and Processing (ICCP 2013)*, pp. 251–258, 2013.
[18] E.P. de Vink, *Designing Stream Based Semantics for Uniform Concurrency and Logic Programming*, Ph.D thesis, Vrije Universiteit Amsterdam, 1990.
[19] Aglets portal site, 2004, `http://aglets.sourceforge.net/`

Computer Science Department, Technical University, Cluj-Napoca, Romania
E-mail address: `eneia.todoran@cs.utcluj.ro`