Continuation Semantics for Asynchronous Concurrency

Gabriel Ciobanu
Institute of Computer Science
Romanian Academy
Iași, Romania
gabriel@info.uaic.ro

Eneia Nicolae Todoran
Department of Computer Science
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
eneia.todoran@cs.utcluj.ro

Abstract. The paper presents a method of reasoning about the behaviour of asynchronous programs in denotational models designed with metric spaces and continuation semantics for concurrency.

Keywords: Metric semantics, continuations for concurrency, asynchronous communication.

1. Introduction

Continuations represent a classical tool in denotational semantics [21]. The distinctive characteristic of the "continuation semantics for concurrency" (CSC) technique [22] is the modelling of continuations as application-specific structures of computations (partially evaluated denotations) rather than the functions to some answer type that are used in the classic technique of continuations [21]. The structure of continuations is representative for the control flow concepts of the concurrent language under study. Unlike other models of concurrency [18, 2, 9], in the CSC approach the final yield of the denotational mapping is a simple collection of observations and all concurrency control concepts are modelled as operations manipulating continuations.

Address for correspondence: gabriel@info.uaic.ro (Gabriel Ciobanu), eneia.todoran@cs.utcluj.ro (Eneia Nicolae Todoran)
We do not know whether the domain of CSC is fully abstract\(^1\). However, it is easy to prove that a denotational semantics designed with CSC is correct with respect to a corresponding operational model [22]. The domain of CSC is general; only the structure of continuations needs to be adapted to the (concurrent) language under study [22, 23, 12]. In this sense the CSC technique provides flexibility in the denotational design of concurrent languages. For example, in the case of a sequential language, a continuation is a stack of computations. It is also natural to use the concept of a multiset to represent parallel computations [23, 12]. For a general combination of sequential and parallel compositions, a continuation is a tree of computations [22].

In this paper we present a continuation semantics for a simple concurrent language \( \mathcal{L} \) embodying a mechanism of asynchronous communication. In theory synchronisation is used because it is simple to express. Based on results expressed in process algebra, asynchronous interaction is primitive, and synchronous communication could be expressed in terms of asynchronous interactions [14]. The relation between synchronous and asynchronous interaction was also investigated in [17]. A simulation of synchronous interaction by means of asynchronous interaction is not always possible. However, asynchronous interaction is easier to implement and it represents a (or rather the) basic communication mechanism in modern distributed computing systems, including most Internet and Web applications.

The language \( \mathcal{L} \) that we consider in this paper embodies the paradigm of asynchronous communication studied in [8]. As explained in [8], this form of asynchronous communication can be encountered in concurrent constraint programming [20], and also in other languages like dataflow, asynchronous CCS and asynchronous CSP. \( \mathcal{L} \) extends the paradigmatic language studied in [8] with recursion. We present a denotational semantics for \( \mathcal{L} \) defined within the mathematical framework of 1-bounded complete metric spaces [2, 1]. The denotational model is designed with the CSC technique.

For the language under consideration, we show that the semantic operators satisfy some concurrency laws, such as the associativity and commutativity of parallel composition. Continuation-based models rely on manipulations of higher-order functions. It may be difficult to reason directly in terms of higher-order functions. Therefore we introduce a left merge operator and we obtain a finite axiomatisation of the parallel composition (or merge) operator. Any non-recursive asynchronous concurrent program is thus provably equivalent to a corresponding nondeterministic sequential program. Obviously, the approach is inspired by classic process algebra theories [16, 6], this approach being adapted by us to a continuation-based framework.

We offer a method of reasoning about the properties of concurrent programs in denotational models designed with CSC. Each semantic property, also called a law here, can be proved by identifying a corresponding invariant of the computation, as a relation between CSC structures. The identification of semantic properties from the invariants of the computation is common in classic bisimulation semantics [16]. Here this idea is adapted to a denotational framework, by using arguments of the kind \( \varepsilon \leq \frac{1}{2} \cdot \varepsilon \Rightarrow \varepsilon = 0 \), which are standard in metric semantics [2]. In our case \( \varepsilon \) is the distance between two behaviourally equivalent continuations, before and after a computation step, respectively. The effect of each computation step is given by the \( \frac{1}{2} \) contracting factor. Therefore \( \varepsilon = 0 \) and the desired property follows. Several proofs are given in the paper. A complete version of the article is available online as a technical report [11].

\(^1\)In fact, we are not aware of any full abstractness result for a concurrent language designed with continuations, although various papers employ continuations in the denotational description of concurrent languages [2, 19, 10].
2. Preliminaries

The notation \((x \in X)\) introduces the set \(X\) with typical element \(x\) ranging over \(X\). Let \(f : X \to Y\) be a function. The function \([ f | x \mapsto y ] : X \to Y\), is defined (for \(x, x' \in X, y \in Y\)) by: \([ f | x \mapsto y ](x') = f(x')\) if \(x' = x\) then \(y\) else \(f(x')\). Instead of \([ [ f [ x_1 \mapsto y_1 ] | x_2 \mapsto y_2 ]\) we often write \([ f [ x_1 \mapsto y_1 ] | x_2 \mapsto y_2 ]\).

If \(f : X \to X\) and \(f(x) = x\) we call \(x\) a fixed point of \(f\). When this fixed point is unique (see Theorem 2.1) we write \(x = \text{fix}(f)\).

The denotational semantics given in this paper is built within the mathematical framework of \(1\)-bounded complete metric spaces (i.e., metric spaces where the distance is less than or equal to 1). We work with the following notions which we assume known: metric and ultrametric space, isometry (distance preserving bijection between metric spaces), complete metric space, and compact set. For a comprehensive presentation of the metric approach to semantics the reader may consult [2].

Examples

1. If \((x, y \in X)\) is any nonempty set, one can define the discrete metric on \(X\) \((d : X \times X \to [0, 1])\) as follows: \(d(x, y) = 0\) if \(x = y\) and \(1\) else \(1\). \((X, d)\) is a complete ultrametric space.

2. A central idea in metric semantics is to state that two computations have distance \(2^{-n}\) whenever the first difference in their behaviours appears after \(n\) computation steps. Let \((a \in A)\) be a nonempty set, and let \((x, y \in A)\) \(A^\infty = A^* \cup A^\omega\), where \(A^*(A^\omega)\) is the set of all finite (infinite) sequences over \(A\). One can define a metric over \(A^\infty\) as follows: \(d(x, y) = 2^{-\sup\{n | x[n] = y[n]\}}\), where \(x[n]\) denotes the prefix of \(x\) of length \(n\), in case length\((x) \geq n\), and \(x\) otherwise (by convention, \(2^{-\infty} = 0\). \(d\) is a Baire-like metric. \((A^\infty, d)\) is a complete ultrametric space.

We recall that if \((X, d_X), (Y, d_Y)\) are metric spaces, a function \(f : X \to Y\) is a contraction if \(\exists c \in \mathbb{R}, 0 \leq c < 1, \forall x_1, x_2 \in X : d_Y(f(x_1), f(x_2)) \leq c \cdot d_X(x_1, x_2)\). In metric semantics it is customary to attach a contracting factor \(c = \frac{1}{2}\) to each computation step. When \(c = 1\) the function \(f\) is called nonexpansive. In what follows we denote the set of all nonexpansive functions from \(X\) to \(Y\) by \(X \to Y\).

At the core of the metric approach to semantics is Banach’s fixed point theorem [5].

Theorem 2.1. (Banach)

Let \((X, d_X)\) be a non-empty complete metric space. Each contraction \(f : X \to X\) has a unique fixed point.

Definition 2.2.

Let \((X, d_X), (Y, d_Y)\) be (ultra) metric spaces. On \((x \in X), (y \in Y)\) \(X \times Y\) (the Cartesian product), \(u, v \in X + Y\) (the disjoint union of \(X\) and \(Y\), which can be defined by: \(X + Y = (\{1\} \times X) \cup (\{2\} \times Y)\)), and \(U, V \in \mathcal{P}(X)\) (the power set of \(X\), i.e. the set of all subsets of \(X\)) one can define the following metrics:

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

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

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

(d) \(d_{X+Y} : (X + Y) \times (X + Y) \to [0, 1], \quad\)
\[ d_{X+Y}(u, v) = \begin{cases} (u, v \in X) & \text{then } d_X(u, v) \\
\text{else} & \text{if } (u, v \in Y) \text{ then } d_Y(u, v) \text{ else } 1, \end{cases} \]

(e) \( d_H : \mathcal{P}(X) \times \mathcal{P}(X) \rightarrow [0, 1], \)

\[ d_H(U, V) = \max\{\sup_{u \in U} d(u, V), \sup_{v \in V} d(v, U)\}, \]

where \( d(u, W) = \inf_{w \in W} d(u, w) \) and by convention \( \sup \emptyset = 0, \inf \emptyset = 1 \) \((d_H \text{ is the Hausdorff metric}).\)

We use the abbreviation \( \mathcal{P}_{\text{neo}}(\cdot) \) to denote the power set of non-empty and compact subsets of \( \cdot \).

**Remark 2.3.** Let \((X, d_X), (Y, d_Y), d_{\frac{1}{2}}X, d_{\frac{1}{2}}X \rightarrow Y, d_X \times Y, d_X + Y \) and \( d_H \) be as in Definition 2.2. In case \( d_X, d_Y \) are ultrametrics, so are \( d_{\frac{1}{2}}X, d_{\frac{1}{2}}X \rightarrow Y, d_X \times Y, d_X + Y \) and \( d_H \). Moreover, if \((X, d_X), (Y, d_Y)\) are complete then \( \frac{1}{2} : X \rightarrow Y, X \rightarrow Y, X \times Y, X + Y \), and \( \mathcal{P}_{\text{neo}}(X) \) (with the metrics defined above) are also complete metric spaces [2].

By \( \mathcal{P}_{\text{finite}}(\cdot) \) we denote the power set of finite subsets of \( \cdot \). In general, the construction \( \mathcal{P}_{\text{finite}}(\cdot) \) does not give rise to a complete space. In our study, we use it to create a structure that we endow with the discrete metric. Any set endowed with the discrete metric is a complete ultrametric space.

Two metric spaces are isometric if there exists an isometry between them. To express that two metric spaces \((X, d_X)\) and \((Y, d_Y)\) are isometric we use the notation \( (X, d_X) \cong (Y, d_Y) \). Also, we often suppress the metrics part in domain definitions and write, e.g., \( \frac{1}{2} : X \) instead of \( (X, d_{\frac{1}{2}}X) \).

### 3. Syntax and Continuation Structure for \( L \)

The syntax of \( L \) is given in BNF in Definition 3.1. The basic components are a set \((a \in) \text{Act}\) of atomic actions and a set \((x \in) \text{RV}\) of recursion variables. There is a special symbol \( \delta \in \text{Act} \), whose behaviour is explained below. \( ; \), \( + \) and \( || \) are operators for sequential, nondeterministic and parallel composition, respectively. \( || \) is also called a merge operator, and \( | | \) is the left merge operator.

**Definition 3.1.** (Syntax of \( L \))

- (a) (Statements) \( s(\in \text{Stat}) ::= a \mid x \mid s \mid s + s \mid s || s \mid || s \mid s \mid s \)
- (b) (Guarded statements) \( g(\in \text{GStat}) ::= a \mid g \mid s \mid g + g \mid g || s \mid || g \mid g \)
- (d) (Declarations) \( (D \in) \text{Decl} = \text{RV} \rightarrow \text{GStat} \)
- (e) (Programs) \( (\rho \in) L = \text{Decl} \times \text{Stat} \)

The meaning of atomic actions is defined by an interpretation function \( I : \text{Act} \rightarrow \Sigma \rightarrow (\{\uparrow\} \cup \Sigma) \), where \( (\sigma \in) \Sigma \) is a set of states. If \( I(a)(\sigma) = \uparrow \) the action \( a \) cannot proceed in state \( \sigma \); its execution is suspended. When all processes are suspended deadlock occurs. Notice that \( I(\delta)(\sigma) = \uparrow, \forall \sigma \in \Sigma \), i.e. the action \( \delta \) suspends in all states. \( L \) incorporates the mechanism of asynchronous communication studied in [8]. As explained in [8], this form of asynchronous communication can be encountered in concurrent constraint programming, and also in other languages like dataflow or asynchronous CSP.

We employ an approach to recursion based on declarations and guarded statements [2]. In a guarded statement each recursive call is preceded by at least one elementary action, which guarantees the fact
that the semantic operators are contracting functions in the present metric setting. For the sake of brevity (and without loss of generality) in what follows we assume a fixed declaration $D \in \text{Decl}$, and all considerations in any given argument refer to this fixed $D$.

For inductive proofs we introduce a complexity measure $\varsigma$ that decreases upon recursive calls, a measure that is well-defined due to our restriction to guarded recursion.

**Definition 3.2.** (Complexity measure) The function $\varsigma : \text{Stat} \to \mathbb{N}$ is given by

$$
\varsigma(a) = 1 \\
\varsigma(x) = 1 + \varsigma(D(x)) \\
\varsigma(s_1 \mathbin{\text{op}} s_2) = 1 + \varsigma(s_1) \quad \text{op} \in \{ ; , \parallel \} \\
\varsigma(s_1 \mathbin{\text{op}} s_2) = 1 + \max\{\varsigma(s_1), \varsigma(s_2)\} \quad \text{op} \in \{ +, \parallel \}
$$

In the CSC approach a continuation is a structured configuration of computations. Continuation structures can be designed by using two abstract concepts: the stack to model sequential composition, and the multiset to model parallel composition. For a general combination of parallel and sequential composition, a continuation is a tree of computations with active computations at the leaves [22]. For example, when the denotation of a program fragment $(s_1 \parallel s_2)$; $s_3$ is computed, the denotations of $s_1$ and $s_2$ become leaves in such a tree and the denotation of $s_3$ becomes an inner node. This behaviour is inspired by the concept of a cactus stack [7], a stack with multiple tops that can be active concurrently. In order to define such domains of trees of computations we employ a (partially ordered) set of identifiers: $(\alpha \in)Id$ is the set of all finite, possibly empty, sequences over $\{1, 2\}$ and $\alpha \preceq \alpha'$ iff $\alpha$ is a prefix of $\alpha'$.

In this paper we use the symbol ‘·’ as a concatenation operator over sequences, hence we can represent any nonempty identifier $\alpha \in Id$ by a finite sequence $\alpha = i_1 \cdot \ldots \cdot i_n$, where $i_1, \ldots, i_n \in \{1, 2\}$. We use the symbol $\lambda$ to represent the empty sequence over $\{1, 2\}$ ($\lambda \in Id$).

**Definition 3.3.**

(a) Let $(\alpha \in)Id = \{1, 2\}^*$ be a set of identifiers, equipped with the following partial ordering:

$\alpha \preceq \alpha'$ iff $\alpha' = \alpha \cdot i_1 \cdot \ldots \cdot i_n$ for $i_1, \ldots, i_n \in \{1, 2\}$, $n \geq 0$. We define $\alpha < \alpha'$ iff $\alpha \leq \alpha'$ and $\alpha \neq \alpha'$. If $A \in \mathcal{P}(Id)$, we denote by $\leq_A$ the restriction of $\leq$ to $A$.

(b) We define a function $\max : \mathcal{P}(Id) \to \mathcal{P}(Id)$ by

$$
\max(A) = \{ \alpha \mid \alpha \text{ is a maximal element of } (A, \leq_A) \}.
$$

**Remark 3.4.** $\lambda \leq \alpha$, for any $\alpha \in Id$, which means that $\lambda$ is the least element of $Id$. Also, when $A \in \mathcal{P}(Id)$, $\alpha$ is a maximal element of $(A, \leq_A)$ if $\alpha \in A$ and $\forall \alpha' \in A : -(\alpha < \alpha')$. The concept of a tree that we use in this paper is taken from set theory, where a tree is a partially ordered set in which the predecessors of each element are well-ordered. A set is well-ordered if it is linearly ordered and every nonempty subset has a least element. A set is linearly ordered if any two elements are comparable. There are several books on set theory that provide formal definitions of these concepts; see, e.g., [15]. Here we only explain the concept of a tree by means of an example.

$(Id, \leq)$ is a partially ordered set, i.e. $\leq$ is a binary relation over $Id$ which is reflexive, transitive and antisymmetric. In this paper we only work with finite trees. If $A \subseteq Id$ is a finite subset of $Id$ then
(A, ≤_A) is a finite tree. For example, let A = {α · 1, α · 2, α · 1 · 1, α · 1 · 2, α · 2 · 1, α · 2 · 2, α · 1 · 1 · 1}, for some α ∈ Id. (A, ≤_A) is a finite tree. The maximal elements of (A, ≤_A) are exactly the leaves of the tree: \( \text{max}(A) = \{α · 1 · 1, α · 1 · 2, α · 2 · 1, α · 2 · 2\} \). The predecessors of each element in A are well-ordered. For example, α · 1 · 1 · 1 > α · 1 · 1 and α · 1 · 1 > α · 1. The set of predecessors of α · 1 · 1 · 1 is {α · 1 · 1, α · 1}, which is linearly ordered, i.e. any two elements in {α · 1 · 1, α · 1} are comparable. In general, α_1 and α_2 are comparable iff α_1 ≤ α_2 or α_2 ≤ α_1. Obviously, every nonempty subset of {α · 1 · 1, α · 1} has a least element. In fact, every finite linearly ordered set is well-ordered.

Let \((x ∈ X)\) be a metric domain, i.e. a complete metric space. We use the following notation:

\[ \|X\| = \mathcal{P}_{\text{finite}}(Id) \times (Id → X) \]

Let α ∈ Id, \((π, θ) ∈ \|X\|\) with π ∈ \(\mathcal{P}_{\text{finite}}(Id)\), θ ∈ Id → X. We define id : \(\|X\| → \mathcal{P}_{\text{finite}}(Id)\), id(π, θ) = π. We also use the following abbreviations:

\[ ((π, θ)(α)) = θ(α) \quad (π, θ) \setminus α = (π \setminus \{α\}, θ) \quad [(π, θ) | α → x] = (π \cup \{α\}, [θ | α → x]) \]

The basic idea is that we treat \((π, θ)\) as a ‘function’ with finite graph \(\{(α, θ(α)) | α ∈ π\}\), thus ignoring the behaviour of θ for any α /∈ π (π is the ‘domain’ of \((π, θ)\)). We use this mathematical structure to represent finite partially ordered bags (or multisets)\(^2\) of computations. The set Id is used to distinguish between multiple occurrences of a computation in such a bag. We endow both sets Id and \(\mathcal{P}_{\text{finite}}(Id)\) with discrete metrics; every set with a discrete metric is a complete ultrametric space. By using the composite metrics given in Definition 2.2, \(\{X\}\) becomes also a metric domain. The operators behave as follows. id(π, θ) returns the collection of identifiers for the valid computations contained in the bag \((π, θ)\), \((π, θ)(α)\) returns the computation with identifier α, \((π, θ) \setminus α\) removes the computation with identifier α, and \([(π, θ) | α → x]\) replaces the computation with identifier α.

We use the same notations (including the operator id and the abbreviations \((\cdot)(α), (\cdot)\setminus α, (\cdot | α → x)\)) when \((x ∈ X)\) is an ordinary set: \(\{X\} = \mathcal{P}_{\text{finite}}(Id) × (Id → X)\); in this case we do not endow \(\{X\}\) with a metric.

### 4. Continuation Semantics for \(L\)

We design a continuation-based denotational semantics for \(L\). As a semantic universe for the final yield of our denotational model we employ a standard linear-time domain \((p ∈ P) = \mathcal{P}_{\text{ncal}}(Σ^* ∪ Σ^* · \{δ\} ∪ Σ^*)\). An element of \(Σ^* · \{δ\}\) is a finite sequence terminated with the symbol δ, which denotes deadlock. We use the symbol λ to represent the empty sequence over Σ. This is a slight abuse of notation since we also use the symbol λ to represent the empty sequence over \{1, 2\}; however it is always clear from the context which is the type of λ (either λ ∈ Id or λ ∈ Σ^*). We view \((q ∈ Σ^* ∪ Σ^* · \{δ\} ∪ Σ^*)\) as a complete ultrametric space by endowing it with the Baire metric (see Section 2). We use the notation \(σ · p = \{σ · q | q ∈ p\}\), for any \(σ ∈ Σ\) and \(p ∈ P\). The type of the denotational semantics \(\|\cdot\|\) for \(L\) is \(\text{Stat} → \text{D}\), where:

\(^2\)We avoid using the notion of a partially ordered multiset which is a more refined structure – see [4], or chapter 16 of [2].
\[
\begin{align*}
\mathbb{D} & \cong \text{Cont} \stackrel{1}{\rightarrow} \Sigma \rightarrow \mathbb{P} \\
(\gamma \in)\text{Cont} & = \text{Id} \times \text{Kont} \\
(\kappa \in)\text{Kont} & = \{ \frac{1}{2} \cdot \mathbb{D} \}
\end{align*}
\]

In the equations given above the sets \(\Sigma, \text{Id}\) (and \(\mathcal{P}_{\text{finite}}(\text{Id})\)) are endowed with the discrete (ultra)metric. The composed metric spaces are built up using the metrics of Definition 2.2. To conclude that such a system of equations has a solution, which is unique up to isometry, we rely on the general method developed in [1]. The solution for \(\mathbb{D}\) is obtained as a complete ultrametric space. In [1], the family of complete (ultra)metric spaces is made into a category \(\mathbb{C}\). It is proved that a generalised form of Banach’s fixed point theorem holds, stating that a functor \(\mathcal{F} : \mathbb{C} \rightarrow \mathbb{C}\) that is\ contracting (in a sense) has a unique fixed point (up to isometry). Intuitively, in the equation above the relevant functor is contracting as a consequence of the fact that the recursive occurrence of \(\mathbb{D}\) is preceded by the \(\frac{1}{2}\) factor.

The construction \(\{ \frac{1}{2} \cdot \mathbb{D} \} = \mathcal{P}_{\text{finite}}(\text{Id}) \times (\text{Id} \rightarrow \frac{1}{2} \cdot \mathbb{D})\) was introduced in Section 3. In the sequel \(\vartheta\) ranges over \(\text{Id} \rightarrow \frac{1}{2} \cdot \mathbb{D}\). We call an element of \(\text{Kont}\) a closed continuation and an element of \(\text{Cont}\) an open continuation. A (closed or open) continuation is a representation of what remains to be computed from the program [21]. A closed continuation \(\kappa \in \text{Kont}\) is a self-contained structure of computations. An open continuation \((\alpha, \kappa) \in \text{Cont}\) behaves like an evaluation context [13] for the denotational mapping \(\llbracket \cdot \rrbracket\). In an expression \(\llbracket s \rrbracket(\alpha, \kappa), \llbracket s \rrbracket\) is the active computation which is evaluated with respect to \((\alpha, \kappa)\). Intuitively, an open continuation \((\alpha, \kappa)\) is a structured configuration of computations which contains a hole, indicating the conceptual position of the active computation. The position of the ‘hole’ is given by the identifier \(\alpha\), which is not an element of \(\text{id}(\kappa)\). \(\llbracket \cdot \rrbracket\) is designed to preserve this invariant property: \(\alpha \not\in \text{id}(\kappa)\) and \(\alpha \in \max(\{\alpha\} \cup \text{id}(\kappa))\), i.e., \(\alpha\) is a leaf in the tree \(\{\alpha\} \cup \text{id}(\kappa)\).

The denotational function \(\llbracket \cdot \rrbracket\) is specified in Definition 4.1 with the aid of a mapping \(kc\), which is called a scheduler. The denotational function maps an open continuation to a program behaviour. After producing an elementary step the denotational function transmits the control to the scheduler. The scheduler receives as parameter a closed continuation \(\kappa\) which it maps to a corresponding program behaviour. If the continuation \(\kappa\) is empty (\(\text{id}(\kappa) = \emptyset\)) the scheduler terminates the computation. Otherwise, the scheduler activates a computation (denotation) selected in a nondeterministic manner from the continuation; it decomposes a closed continuation into a computation and a corresponding open continuation and then it executes the former with the latter as continuation.

The semantics of nondeterministic choice in \(\mathcal{L}\) is given by the operator \(+ : (\mathbb{P} \times \mathbb{P}) \rightarrow \mathbb{P}\) defined by \(p_1 + p_2 = \{ q \mid q \in p_1 \cup p_2, q \neq \delta \} \cup \{ \delta \mid \delta \in p_1 \cap p_2 \}\). This definition reflects that \(p_1 + p_2\) blocks only if both \(p_1\) and \(p_2\) block. It is easy to check that + is well-defined, nonexpansive, associative, commutative and idempotent.

**Definition 4.1.** (Denotational semantics for \(\mathcal{L}\))

(a) Let \(kc : \text{Kont} \rightarrow \Sigma \rightarrow \mathbb{P}\) be given by:

\[
kc(\kappa)(\sigma) = \begin{cases} 
\{\lambda\} & \text{if } (\text{id}(\kappa) = \emptyset) \\
+_{\alpha \in \text{max}(\text{id}(\kappa))} \kappa(\alpha)(\kappa \setminus \alpha)(\sigma) & \text{else}
\end{cases}
\]
(b) Let \((\phi \in)\text{Sem} = \text{Stat} \rightarrow D\). We define \(\Phi : \text{Sem} \rightarrow \text{Sem}\) by:

\[
\begin{align*}
\Phi(\phi)(a)(\alpha, \kappa)(\sigma) &= \begin{cases} 
(I(a)(\sigma) = \top) \text{ then } \delta \text{ else } I(a)(\sigma) \cdot kc(\kappa)I(a)(\sigma) 
\end{cases} \\
\Phi(\phi)(x)(\alpha, \kappa)(\sigma) &= \Phi(\phi)(D(x))(\alpha, \kappa)(\sigma) \\
\Phi(\phi)(s_1; s_2)(\alpha, \kappa)(\sigma) &= \Phi(\phi)(s_1)(\alpha \cdot 1, [\kappa \mid \alpha \rightarrow \phi(s_2)])(\sigma) \\
\Phi(\phi)(s_1 + s_2)(\alpha, \kappa)(\sigma) &= \Phi(\phi)(s_1)(\alpha, \kappa)(\sigma) + \Phi(\phi)(s_2)(\alpha, \kappa)(\sigma) \\
\Phi(\phi)(s_1 \parallel s_2)(\alpha, \kappa)(\sigma) &= \Phi(\phi)(s_1)(\alpha \cdot 1, [\kappa \mid \alpha \cdot 2 \rightarrow \phi(s_2)])(\sigma) \\
\Phi(\phi)(s_1 \mid s_2)(\alpha, \kappa)(\sigma) &= \Phi(\phi)(s_1)(\alpha \cdot 1, [\kappa \mid \alpha \cdot 2 \rightarrow \phi(s_2)])(\sigma) + \Phi(\phi)(s_2)(\alpha \cdot 1, [\kappa \mid \alpha \cdot 2 \rightarrow \phi(s_1)])(\sigma)
\end{align*}
\]

(c) We put \([\cdot] = \text{fix}(\Phi)\). Let \(\alpha_0 \equiv \lambda\), and \(\kappa_0 = (\emptyset, \emptyset)\), where \(\emptyset(\alpha) = [\delta], \forall \alpha \in \text{Id.}\ (\alpha_0, \kappa_0) \in \text{Cont}\) is the empty continuation. We define \(D[\cdot] : \text{Stat} \rightarrow \Sigma \rightarrow \text{P}\) by:

\[
D[s] = [s](\alpha_0, \kappa_0)
\]

The semantics of atomic actions is defined with the aid of the interpretation function \(I\) introduced in the paragraph that follows after Definition 3.1.

A continuation is a tree of computations with active elements at the leaves (the maximal elements with respect to \(\leq\)). In the case of a sequential composition \((s_1; s_2)\) the computations \([s_1]\) and \([s_2]\) are given the identifiers \(\alpha \cdot 1\) and \(\alpha\), respectively (with \(\alpha \cdot 1 > \alpha\)). The scheduler function \(kc\) gives priority to the computations at the leaves of the tree that represents the continuation. Hence \([s_2]\) will only be evaluated after the completion of the evaluation of \([s_1]\). In the case of a parallel composition \((s_1 \parallel s_2)\) the computations \([s_1]\) and \([s_2]\) are given identifiers \((\alpha \cdot 1\) and \(\alpha \cdot 2)\) that are incomparable (with respect to \(\leq\)) hence the computations \([s_1]\) and \([s_2]\) are evaluated in an interleaved manner. The parallel composition of two statements is implemented based on a nondeterministic choice between two alternative computations: one starting from the first statement and one starting from the second statement. Hence, all possible interleavings are obtained.

The denotational semantics \([\cdot]\) is defined as the (unique) fixed point of \(\Phi\). It may not be obvious why on the right-hand sides of the equations given in Definition 4.1(b), in some places we use \(\Phi(\phi)\) while in others we use \(\phi\). The definition of \(\Phi(\phi)\) is organised by induction on \(\varsigma(s)\) (see Definition 3.2). The computations \(\phi(s)\) only occur in the continuation and are always executed after an elementary step (performed by the active computation) which ensures the contractiveness of \(\Phi\). Definition 4.1 is justified by Lemma 4.2 and Lemma 4.3, whose proofs are omitted. Similar lemmas are given in [22]. In Lemma 4.3(b), \(\Phi(\phi)(s)\) is (only) nonexpansive (rather than contractive) in the continuation. Still, this implies that \(\Phi\) is \(\frac{1}{2}\) - contractive in \(\phi\). This is a consequence of the \(\frac{1}{2}\) - contracting factor in our domain equation. Intuitively, the distance between denotations halves (only) while they are stored into a continuation. This also explains the occurrence of the multiplication factor \(\cdot\) (rather than \(\frac{1}{2}\) - ) in Lemma 4.2(b).

**Lemma 4.2.**

(a) The mapping \(kc\) (introduced in Definition 4.1(a)) is well-defined.

(b) \(\forall \kappa_1, \kappa_2 \in \text{Kont} : d(kc(\kappa_1), kc(\kappa_2)) \leq 2 \cdot d(\kappa_1, \kappa_2)\)
Lemma 4.3. For all $\phi \in (\text{Stat} \to D), s \in \text{Stat}, \alpha \in \text{Id}, \kappa \in \text{Kont}, \sigma \in \Sigma$:

(a) $\Phi(\phi)(s)(\alpha, \kappa)(\sigma) \in \mathcal{P}$ (it is well defined),

(b) $\Phi(\phi)(s)$ is nonexpansive (in $(\alpha, \kappa)$), and

(c) $\Phi$ is $\frac{1}{2}$-contractive (in $\phi$).

5. Concurrency Laws

We present a method of describing the behaviour of concurrent systems in a denotational model designed with CSC, using a representation of continuations as structured configurations of computations. For the language $L$ we show that the semantic operators satisfy laws that are usually included in concurrency theories, such as the associativity and commutativity of parallel composition.

Various properties can be proved for all continuations by simple manipulations of the semantic equations. However, the flexibility provided by continuations comes at a price. Some properties may require more complex arguments and can be obtained for continuations that contain only denotations of program statements. We introduce the auxiliary notion of a configuration and a notion of isomorphism over configurations. A configuration is a structure of $L$ statements. A continuation may contain arbitrary values of the type $D$. We prove the desired properties for continuations that can be obtained semantically as versions of configurations, i.e. for continuations that contain only denotations of statements. This represents an invariant of the denotational semantics, and ensures its consistency just because the initial continuation is empty and the denotational semantics adds to the continuation only denotations of statements. The function $K$ defined in Definition 5.1(b) maps a configuration to a corresponding continuation.

Definition 5.1.

(a) We define the set of closed configurations $(k \in)\text{Konf} = \langle \text{Stat} \rangle$. The construct $\langle \cdot \rangle$ is used here to define an ordinary set; see the explanation given in the final part of Section 3. Also, we define the set $\text{Conf}$ of open configurations by:

$$\text{Conf} = \{(\alpha, k) \mid (\alpha, k) \in (\text{Id} \times \text{Konf}), \alpha \notin \text{id}(k), \alpha \in \max(\{\alpha\} \cup \text{id}(k))\}$$

(b) We define $K : \text{Konf} \to \text{Kont}$, $K(k) = (\text{id}(k), \emptyset)$, where $\emptyset(\alpha) = [k(\alpha)], \forall \alpha \in \text{Id}$.

Definition 5.2.

(a) We say that two closed configurations $k_1, k_2 \in \text{Konf}$ are isomorphic, and we write $k_1 \cong k_2$, iff there exists a bijection $\mu : \text{id}(k_1) \to \text{id}(k_2)$ such that:

(i) $\mu(\alpha') \leq \mu(\alpha'') \iff \alpha' \leq \alpha'', \forall \alpha', \alpha'' \in \text{id}(k_1)$

(ii) $k_2(\mu(\alpha)) = k_1(\alpha), \forall \alpha \in \text{id}(k_1)$

(b) We say that two open configurations $(\alpha_1, k_1), (\alpha_2, k_2) \in \text{Conf}$ are isomorphic, and write $(\alpha_1, k_1) \cong (\alpha_2, k_2)$ iff there exists a bijection $\mu : (\{\alpha_1\} \cup \text{id}(k_1)) \to (\{\alpha_2\} \cup \text{id}(k_2))$ such that:

(i) $\mu(\alpha_1) = \alpha_2$
Obviously, \( \forall (\alpha, k) \in \text{Conf} \) : \( (\alpha, k) \cong (\alpha, k) \) and if \( (\alpha_1, k_1), (\alpha_2, k_2) \in \text{Conf} \) then \( (\alpha_1, k_1) \cong (\alpha_2, k_2) \Rightarrow k_1 \cong k_2 \). Also, the following lemma is easily established.

**Lemma 5.3.** \( \forall k \in \text{Konf}, \alpha \in \text{Id}, s \in \text{Stat} : K[k \mid \alpha \mapsto s] = [K(k) \mid \alpha \mapsto [s]] \).

In Corollary 5.5 we show that any two continuations that correspond to isomorphic configurations behave the same, by combining Lemma 5.4 with an argument ‘\( \varepsilon \leq \frac{1}{2} \cdot \varepsilon \Rightarrow \varepsilon = 0 \)’. Lemma 5.4 identifies the property - in this case the isomorphism between configurations - that is preserved by each computation step. The effect of each computation step is given by the \( \frac{1}{2} \)-contracting factor.

**Lemma 5.4.**

(a) For all \( k_1, k_2 \in \text{Konf} \) with \( k_1 \cong k_2 \) and \( \sigma \in \Sigma \), there exists \( \overline{\sigma} \in \text{Stat}, (\overline{\sigma}_1, \overline{K}_1), (\overline{\sigma}_2, \overline{K}_2) \in \text{Conf} \) with \( (\overline{\sigma}_1, \overline{K}_1) \cong (\overline{\sigma}_2, \overline{K}_2) \) and \( \sigma \in \Sigma \) such that:

\[
d(kc(K(k_1)))(\sigma), kc(K(k_2))(\sigma)) \leq d([\overline{\sigma}](\overline{\sigma}_1, K(\overline{K}_1))(\sigma), [\overline{\sigma}](\overline{\sigma}_2, K(\overline{K}_2))(\sigma))
\]

(b) For all \( s \in \text{Stat}, (\alpha_1, k_1), (\alpha_2, k_2) \in \text{Conf} \) with \( (\alpha_1, k_1) \cong (\alpha_2, k_2) \) and \( \sigma \in \Sigma \), there exists \( \overline{\sigma} \in \text{Stat}, (\overline{\sigma}_1, \overline{K}_1), (\overline{\sigma}_2, \overline{K}_2) \in \text{Conf} \) with \( (\overline{\sigma}_1, \overline{K}_1) \cong (\overline{\sigma}_2, \overline{K}_2) \) and \( \sigma \in \Sigma \) such that:

\[
d([s](\alpha_1, K(k_1))(\sigma), [s](\alpha_2, K(k_2))(\sigma)) \leq \frac{1}{2} \cdot d([\overline{\sigma}](\overline{\sigma}_1, K(\overline{K}_1))(\sigma), [\overline{\sigma}](\overline{\sigma}_2, K(\overline{K}_2))(\sigma))
\]

**Proof:**

We only treat Lemma 5.4(b) by induction on \( \varsigma(s) \). One subcase: \( s = s_1 \parallel s_2 \).

\[
d([s_1 \parallel s_2](\alpha_1, K(k_1))(\sigma), [s_1 \parallel s_2](\alpha_2, K(k_2))(\sigma))
\]

\[
= d([s_1](\alpha_1 \cdot 1, [K(k_1) \mid \alpha_1 \cdot 2 \mapsto [s_2]](\sigma)) + [s_2](\alpha_1 \cdot 1, [K(k_1) \mid \alpha_1 \cdot 2 \mapsto [s_1]](\sigma)),
\]

\[
    [s_1](\alpha_2 \cdot 1, [K(k_2) \mid \alpha_2 \cdot 2 \mapsto [s_2]](\sigma)) + [s_2](\alpha_2 \cdot 1, [K(k_2) \mid \alpha_2 \cdot 2 \mapsto [s_1]](\sigma))
\]

[Lemma 5.3, + is nonexpansive]

\[
\leq \max\{d([s_1](\alpha_1 \cdot 1, K[k_1 \mid \alpha_1 \cdot 2 \mapsto s_2])(\sigma), [s_1](\alpha_2 \cdot 1, K[k_2 \mid \alpha_2 \cdot 2 \mapsto s_2])(\sigma)),
\]

\[
d([s_2](\alpha_1 \cdot 1, K[k_1 \mid \alpha_1 \cdot 2 \mapsto s_1])(\sigma), [s_2](\alpha_2 \cdot 1, K[k_2 \mid \alpha_2 \cdot 2 \mapsto s_1])(\sigma))\}
\]

For \((*)\) it is easy to check that \( (\alpha_1, k_1) \cong (\alpha_2, k_2) \) implies \( (\alpha_1 \cdot 1, [k_1 \mid \alpha_1 \cdot 2 \mapsto s_2]) \cong (\alpha_2 \cdot 1, [k_2 \mid \alpha_2 \cdot 2 \mapsto s_2]) \). For example, one can use a bijection that maps \( \alpha_1 \cdot 1 \) to \( \alpha_2 \cdot 1 \) and \( \alpha_1 \cdot 2 \) to \( \alpha_2 \cdot 2 \). As \( \varsigma(s_1) < \varsigma(s_1 \parallel s_2) \) we can use the induction hypothesis for \((*)\) and we infer that \( \exists \sigma \in \text{Stat}, (\overline{\sigma}_1, \overline{K}_1) \cong (\overline{\sigma}_2, \overline{K}_2) \in \text{Conf} \) and \( \sigma \in \Sigma \) such that:

\[
(*) \leq \frac{1}{2} \cdot d([\overline{\sigma}](\overline{\sigma}_1, K(\overline{K}_1))(\sigma), [\overline{\sigma}](\overline{\sigma}_2, K(\overline{K}_2))(\sigma))
\]

\((**\) can be handled in a similar manner and the desired result follows immediately. \(\square\)
Corollary 5.5.

(a) For all \( s \in \text{Stat}, (\alpha_1, k_1) \cong (\alpha_2, k_2) \in \text{Conf} \): \( [s](\alpha_1, K(k_1)) = [s](\alpha_2, K(k_2)) \).
(b) For all \( k_1 \cong k_2 \in \text{Konf} \): \( kc(K(k_1)) = kc(K(k_2)) \).

Proof:

Let

\[
(w \in) W_D = \{ (s, (\alpha_1, k_1), (\alpha_2, k_2), \sigma) \mid s \in \text{Stat}, (\alpha_1, k_1), (\alpha_2, k_2) \in \text{Conf} : (\alpha_1, k_1) \cong (\alpha_2, k_2), \sigma \in \Sigma \}
\]

For \( (s, (\alpha_1, k_1), (\alpha_2, k_2), \sigma) \in W_D \) we use the notation:

\[
\varepsilon_D(s, (\alpha_1, k_1), (\alpha_2, k_2), \sigma) \not{=} d([s](\alpha_1, K(k_1))(\sigma), [s](\alpha_2, K(k_2))(\sigma))
\]

Let \( (s, (\alpha_1, k_1), (\alpha_2, k_2), \sigma) \in W_D \). By Lemma 5.4(b) there exists \( (\pi, (\pi_1, k_1), (\pi_2, k_2), \pi) \in W_D \) such that: \( \varepsilon_D(s, (\alpha_1, k_1), (\alpha_2, k_2), \sigma) \leq \frac{1}{2} \cdot \varepsilon_D(\pi, (\pi_1, k_1), (\pi_2, k_2), \pi) \). Hence

\[
\sup_{w \in W_D} \varepsilon_D(w) \leq \frac{1}{2} \cdot \sup_{\pi \in W_D} \varepsilon_D(\pi)
\]

where \( w = (s, (\alpha_1, k_1), (\alpha_2, k_2), \sigma), \pi = (\pi, (\pi_1, k_1), (\pi_2, k_2), \pi) \). Therefore \( \sup_{w \in W_D} \varepsilon_D(w) = 0 \), i.e. \( d([s](\alpha_1, K(k_1))(\sigma), [s](\alpha_2, K(k_2))(\sigma)) = 0 \). Hence \( [s](\alpha_1, K(k_1))(\sigma) = [s](\alpha_2, K(k_2))(\sigma) \) for any \( \sigma \in \Sigma \), which implies Corollary 5.5(a). Corollary 5.5(b) follows immediately from Lemma 5.4(a) and Corollary 5.5(a).

We show that CSC can be used to reason in a compositional manner upon the behaviour of concurrent programs. For this purpose we introduce a notion of syntactic context for the class of \( L \) statements.

Definition 5.6. (Contexts for \( L \))

\[
C ::= \cdot | a | x | C; C | C + C | C \parallel C | C||C
\]

We denote by \( C(s) \) the result of substituting \( s \) for all occurrences of \( \cdot \) in \( C \). This substitution can be defined inductively: \( (\cdot)(s) = s, a(s) = a, x(s) = x \) and \( (C_1 \text{ op } C_2)(s) = C_1(s) \text{ op } C_2(s) \), where \( \text{ op } \in \{;+, \|, \||\} \).

Lemma 5.7 shows that program properties are preserved in any syntactic context by all continuations containing only denotations of statements; its proof is given in [11]. The proof involves again the identification of an appropriate computing invariant and the use of contraction \( \varepsilon \leq \frac{1}{2} \cdot \varepsilon \Rightarrow \varepsilon = 0 \).

Lemma 5.7. If \( s_1, s_2 \in \text{Stat} \) are such that for all \( (\alpha, k) \in \text{Conf} \):

\[
[s_1](\alpha, K(k)) = [s_2](\alpha, K(k))
\]

then for all \( (\alpha, k) \in \text{Conf} \) and for all contexts \( C \):

\[
[C(s_1)](\alpha, K(k)) = [C(s_2)](\alpha, K(k))
\]
Theorem 5.10 presents the main results for \( L \). It allows us to reason in a compositional manner upon the behaviour of \( L \) asynchronous programs. The denotational semantics \([\cdot]\) preserves the following invariant property: continuations contain only computations denotable by program statements. The initial continuation \( (\alpha_0, \kappa_0) \) (see Definition 4.1(c)) is empty (contains no computations) and each equation in the definition of \([\cdot]\) adds only denotations of statements to the continuation. The properties given in Theorem 5.10 hold for continuations containing only computations denotable by program statements, which is sufficient in practice.

The proof of Theorem 5.10 relies on Lemma 5.8 and Lemma 5.9. Essentially, Lemma 5.8 and Lemma 5.9 identify (non-isomorphic) continuation structures - specific of sequential composition and parallel composition - respectively - that behave the same; these lemmas can be approached with the same method that was used in the proof of Lemma 5.4 and Corollary 5.5, by the identification of appropriate invariant properties in combination with uses of the ‘\( \varepsilon \leq \frac{1}{2} \cdot \varepsilon \Rightarrow \varepsilon = 0 \)’ argument. The proofs of Lemma 5.8 and 5.9 are given in the published technical report [11].

**Lemma 5.8.** For all \( \pi, s_1, s_2 \in \text{Stat}, \sigma \in \Sigma, \pi, \alpha \in Id, k \in \text{Konf} \) such that \( (\pi, k) \in \text{Conf}, \alpha \notin \text{id}(k), \alpha \cdot 1 \notin \text{id}(k) \) and \( \sim(\pi \leq \alpha \cdot 1) \) we have:
(a) \( kc[K(k) | \alpha \mapsto [s_1; s_2]](\sigma) = kc[K(k) | \alpha \cdot 1 \mapsto [s_1] | \alpha \mapsto [s_2]](\sigma) \)
(b) \( K(\pi, [K(k) | \alpha \mapsto [s_1; s_2]])(\sigma) = K(\pi, [K(k) | \alpha \cdot 1 \mapsto [s_1] | \alpha \mapsto [s_2]])(\sigma) \)

**Lemma 5.9.** For all \( \pi, s_1, s_2 \in \text{Stat}, \sigma \in \Sigma, \pi, \alpha \in Id, k \in \text{Konf} \) such that \( (\pi, k) \in \text{Conf}, (\alpha, k) \in \text{Conf}, \sim(\pi \leq \alpha) \) and \( \sim(\alpha \leq \pi) \) we have:
(a) \( kc[K(k) | \alpha \mapsto [s_1] [s_2]](\sigma) = kc[K(k) | \alpha \cdot 1 \mapsto [s_1] | \alpha \cdot 2 \mapsto [s_2]](\sigma) \)
(b) \( K(\pi, [K(k) | \alpha \mapsto [s_1] [s_2]])(\sigma) = K(\pi, [K(k) | \alpha \cdot 1 \mapsto [s_1] | \alpha \cdot 2 \mapsto [s_2]])(\sigma) \)

In the sequel we write \( s \simeq \tilde{s} \) (\( s, \tilde{s} \in \text{Stat} \)) to express that \( [C(s)](\alpha, K(k)) = [C(\tilde{s})](\tilde{\pi}, K(\tilde{k})) \) for all contexts \( C \) and for all isomorphic configurations \( (\alpha, k) \simeq (\tilde{\pi}, \tilde{k}) \) (\( \in \text{Conf} \)).

**Theorem 5.10.** For all \( s, s_1, s_2, s_3 \in \text{Stat} : \\
(a) \ s_1 + s_2 \simeq s_2 + s_1 \ \ (\text{commutativity of } +)  \\
(b) \ (s_1 + s_2) + s_3 \simeq s_1 + (s_2 + s_3) \ \ (\text{associativity of } +)  \\
(c) \ s + s \simeq s \ \ (\text{idempotency of } +)  \\
(d) \ (s_1 + s_2); s_3 \simeq s_1; s_3 + s_2; s_3 \ \ (\text{right distributivity of } ; \overline{\text{over }} +)  \\
(e) \ s_1; (s_2; s_3) \simeq (s_1; s_2); s_3 \ \ (\text{associativity of } ;)  \\
(f) \ s + \delta \simeq s  \\
(g) \ \delta s \simeq \delta  \\
(h) \ s_1 \parallel s_2 \simeq s_1 \parallel s_2 \parallel s_1 \ \ (\text{right distributivity of } \parallel \text{ over } +)  \\
(i) \ a \parallel s \simeq a; s  \\
(j) \ (a; s_1) \parallel s_2 \simeq a; (s_1 \parallel s_2)  \\
(k) \ (s_1 + s_2) \parallel s_3 \simeq s_1 \parallel s_3 + s_2 \parallel s_3 \ \ (\text{right distributivity of } \parallel \text{ over } +)  \\
(l) \ s_1 \parallel s_2 \simeq s_2 \parallel s_1 \ \ (\text{commutativity of } \parallel)  \\
(m) \ s_1 \parallel (s_2 \parallel s_3) \simeq (s_1 \parallel s_2) \parallel s_3 \ \ (\text{associativity of } \parallel) \\

Similarly where $\alpha$ which implies $[ s_1 + s_2 ](\alpha, \kappa)(\sigma) = [ s_2 ](\alpha, \kappa)(\sigma) + [ s_1 ](\alpha, \kappa)(\sigma)$ $[ + \text{ is commutative} ]$

Next, $[ s ] = [ \sigma ] \Rightarrow s \simeq \sigma, \forall s, \sigma \in \text{Stat}$. Indeed, $[ s ] = [ \sigma ] \Leftrightarrow [ C(s) ] = [ C(\sigma) ]$ for any context $C$ (by the compositionality of $[ . ]$ and $[ C(s) ] = [ C(\sigma) ]$). Hence $s_1 + s_2 \simeq s_2 + s_1$. The properties stated by Theorem 5.10 (a)-(d), (f)-(h) and (k) can all be handled in this way. Property 5.10(i) is an easy consequence of properties 5.10(h) and (a).

The properties stated by Theorem 5.10(e), (i), (j) and (m) can be proved for continuations containing only denotations of statements (not for arbitrary continuations) and require more involved arguments based on the identification of computing invariants and the use of contraction. A complete proof of Theorem 5.10 is given in [11]. Property 5.10(e) can be proved by using Lemma 5.8. Property 5.10(i) follows by using Corollary 5.5. In the proofs of Theorem 5.10(j) and (m) one can use Lemma 5.9. Here we only treat Theorem 5.10(m). By Lemma 5.7 and Corollary 5.5(a), in order to prove $s \simeq \sigma$ it is enough to show that $[ s ](\alpha, K(k)) = [ \sigma ](\alpha, K(k))$ for any $\alpha, k \in \text{Conf}$. We prove that

$$[ s_1 \| (s_2 \| s_3) ](\alpha, K(k))(\sigma) = [ (s_1 \| s_2) \| s_3 ](\alpha, K(k))(\sigma)$$

for any $\alpha, k \in \text{Conf}, \sigma \in \Sigma$.

where $\alpha_1 = \alpha \cdot 1$, $\alpha_2 = \alpha_3 = \alpha \cdot 1 \cdot 1$ and

$$\kappa_1 = [ K(k) \mid \alpha \cdot 2 \mapsto [ s_2 \| s_3 ] ]$$

$$\kappa_2 = [ K(k) \mid \alpha \cdot 2 \mapsto [ s_1 ] \| \alpha \cdot 1 \cdot 2 \mapsto [ s_3 ] ]$$

$$\kappa_3 = [ K(k) \mid \alpha \cdot 2 \mapsto [ s_1 ] \| \alpha \cdot 1 \cdot 2 \mapsto [ s_2 ] ]$$

Similarly

$$[ (s_1 \| s_2) \| s_3 ](\alpha, K(k))(\sigma) = [ s_1 ](\pi_1, \pi_1)(\sigma) + [ s_2 ](\pi_2, \pi_3)(\sigma) + [ s_3 ](\pi_3, \pi_3)(\sigma)$$

where $\overline{\pi_1} = \overline{\pi_2} = \alpha \cdot 1 \cdot 1$, $\overline{\pi_3} = \alpha \cdot 1$ and

$$\pi_1 = [ K(k) \mid \alpha \cdot 2 \mapsto [ s_3 ] \| \alpha \cdot 1 \cdot 2 \mapsto [ s_2 ] ]$$

$$\pi_2 = [ K(k) \mid \alpha \cdot 2 \mapsto [ s_3 ] \| \alpha \cdot 1 \cdot 2 \mapsto [ s_1 ] ]$$

$$\pi_3 = [ K(k) \mid \alpha \cdot 2 \mapsto [ s_1 ] \| s_2 ]$$
In order to obtain the desired result it is enough to show that \([s_1](\alpha_i, \kappa_i)(\sigma) = [s_1](\alpha_i, \pi_i)(\sigma)\), for
\(i = 1, 2, 3\). The proofs are very similar. Here we only show that \([s_1](\alpha_1, \kappa_1)(\sigma) = [s_1](\pi_1, \pi_1)(\sigma)\).

\[
[s_1](\alpha \cdot 1, [K(k) | \alpha \cdot 2 \mapsto [s_2 || s_3]])(\sigma) \quad \text{[Lemma 5.9(b)]}
\]

\[
= [s_1](\alpha \cdot 1, [K(k) | \alpha \cdot 2 \mapsto [s_2] | \alpha \cdot 2 \mapsto [s_3]])(\sigma) \quad \text{[Lemma 5.3]}
\]

\[
= [s_1](\alpha \cdot 1, [K[k | \alpha \cdot 2 \mapsto s_2 | \alpha \cdot 2 \mapsto s_3]])(\sigma)
\]

It is easy to check that \((\alpha, k) \in Conf\) implies

\[(\alpha \cdot 1, [k \mid \alpha \cdot 2 \mapsto s_2 | \alpha \cdot 2 \mapsto s_3]) \equiv (\alpha \cdot 1, [k \mid \alpha \cdot 2 \mapsto s_3 | \alpha \cdot 1 \cdot 2 \mapsto s_2]).\]

Hence

\[
[s_1](\alpha \cdot 1, K[k | \alpha \cdot 2 \mapsto s_2 | \alpha \cdot 2 \mapsto s_3])(\sigma) \quad \text{[Corollary 5.5(a)]}
\]

\[
= [s_1](\alpha \cdot 1, K[k | s_3 | \alpha \cdot 1 \cdot 2 \mapsto s_2])(\sigma) \quad \text{[Lemma 5.3]}
\]

\[
= [s_1](\alpha \cdot 1, [K(k) | \alpha \cdot 2 \mapsto [s_3] | \alpha \cdot 1 \cdot 2 \mapsto [s_2]])(\sigma)
\]

\[\square\]

**Remark 5.11.** As it is well-known, the properties stated by Theorem 5.10 provide a finite axiomatisation for the parallel composition operator \(||\); see, e.g., [3]. For any non-recursive \(L\) program (closed term) \(s \in Stat\) there is a non-recursive program \(\pi \in Stat\) that contains only the operators for sequential composition (;) and nondeterministic choice (+) and such that the above set of laws implies \(s \simeq \pi\). The operators \(||\) and \(\|\) can be eliminated from any non-recursive asynchronous \(L\) program. **Such an elimination can always be performed without manipulating continuations explicitly.** For example \(a_1 || a_2 \simeq a_1 \| a_2 + a_2\| a_1\).

Recall that \(D[s][\sigma] = [s](\alpha_0, \kappa_0)(\sigma)\) (see Definition 4.1(c)). For any \(a \in Act, s, s' \in Stat\) and for any context \(C\) one can easily check the following:

- \(s \simeq s'\) implies \(D[C(s)] = D[C(s')]\).
- \(D[a; s][\sigma] = \text{if } (I(a)(\sigma) = \uparrow) \text{ then } \{\delta\} \text{ else } I(a)(\sigma) \cdot D[s][\delta](I(a)(\sigma))\)
- \(D[s + s'][\sigma] = D[s][\sigma] + D[s'][\sigma]\).

For example, if \(I(a_1)(\sigma) = \sigma_1, I(a_2)(\sigma_1) = \sigma_2\) and \(I(a_2)(\sigma) = \uparrow\) then

\[
D[a_1||a_2][\sigma] = D[a_1||a_2 + a_2||a_1][\sigma] = D[a_1; a_2 + a_2; a_1][\sigma] = D[a_1; a_2][\sigma] + D[a_2; a_1][\sigma] = \{\sigma_1 \sigma_2\} + \{\delta\} = \{\sigma_1 \sigma_2\}.
\]

\[\text{Strictly speaking, properties given in Theorem 5.10(l) and 5.10(m) are not needed for this purpose.}\]

\[\text{Notice that } \kappa_0 = K(k_0), \text{ where } k_0 = (0, t_0), \text{ with } t_0 \in (Id \rightarrow Stat), t_0(\alpha) = \delta, \forall \alpha \in Id. \text{ The property follows by using Corollary 5.5 and the fact that } [k_0 \mid \alpha' \mapsto s] \\backslash \alpha' \simeq k_0, \forall \alpha' \in Id. \text{ In fact, } [k \mid \alpha' \mapsto s] \\backslash \alpha' \simeq k, \forall k \in Konf, \text{ if } \alpha' \notin id(k).\]
Remark 5.12. In the continuation semantics given in this paper computations proceed according to the concurrency laws that are well-known from classic process algebra theories [3]. However, a model identifying the behaviour of all computations would also satisfy these laws. A general result concerning the ability of our continuations semantics to distinguish between computations that should be distinguished is beyond the scope of this paper. We only consider here a significant example.

The semantics of sequential composition (;) is defined using continuations, but it ultimately relies on the prefixing operation σ·p. Notice that if δ /∈ p then σ·(p+{δ}) = σ·p ≠ σ·p+{σδ} = σ·p + σ·{δ}. Hence, we cannot expect to get a model which also satisfies the law \( s_1; (s_2 + s_3) \simeq s_1; s_2 + s_1; s_3 \) (left distributivity of ; over +). For example, if \( σ, σ_1 ∈ Σ, I(a_1)(σ) = σ_1 \) and \( I(a_2)(σ_1) = σ_2 \) then

\[
D[a_1; (a_2 + δ)]σ = σ_1 · (σ_2 + δ)] = \neq (σ_1σ_2) = D[a_1; a_2 + a_1; δ]σ.
\]

Also, it is easy to check that \( D[s_1] \neq D[s_2] \Rightarrow s_1 \neq s_2 \) and \( s_1 \neq s_2 \Rightarrow [s_1] \neq [s_2] \), where we write \( s_1 \neq s_2 \) to express that \( ¬(s_1 \simeq s_2) \).

6. Concluding Remarks

We presented a method of describing the behaviour of programs in denotational models designed with metric spaces and continuation semantics for concurrency (CSC). The method was illustrated by designing a continuation semantics for a simple asynchronous language. We proved that the semantic operators designed with continuations obey concurrency laws such as the associativity and commutativity of parallel composition. The method is general; we think it could be applied to every language designed by using CSC. The method relies on the identification of computing invariants as relations between continuations in combination with arguments of the kind ‘\( ε ≤ \frac{1}{2} · ε \Rightarrow ε = 0 \)’, which are standard in metric semantics. The significance of the results is given by the flexibility provided by the continuations technique which can thus be used to describe concurrent behaviour.

In this paper the domain of continuations (\( Kont = \{ \frac{1}{2} · D \} \)) was modelled with the aid of a function space from a set of identifiers (endowed with a partial order) to the domain of computations: \( Id → \frac{1}{2} · D \) (see Section 3 where the construction \( [·] \) was introduced). According to Corollary 5.5, any two isomorphic continuations behave the same. Intuitively, the domain of continuations could be defined in terms of isomorphism classes \( [Id → \frac{1}{2} · D] \) of such structures. Since the existing models based on isomorphism classes of semantic structures (in particular the metric pomset model [4]) do not involve domains defined by reflexive equations (like \( D \)), such a construction also requires further work.

References


