Electronic Notes in Theoretical Computer Science 40 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume40.html 44 pages

Linear Structures for Concurrency in Probabilistic Programming Languages

Alessandra Di Pierro

Dipartimento di Informatica, Universitá di Pisa, Italy

Herbert Wiklicky

Department of Computing, Imperial College, London, UK

Abstract

We introduce a semantical model based on operator algebras and we show the suitability of this model to capture both a quantitative version of non-determinism (in the form of a probabilistic choice) and concurrency. We present the model by referring to a generic language which generalises various probabilistic concurrent languages from different programming paradigms. We discuss the relation between concurrency and the commutativity of the resulting semantical domain. In particular, we use Gelfand's representation theorem to relate the semantical models of synchronisation-free and fully concurrent versions of the language. A central aspect of the model we present is that it allows for a unified view of both operational and denotational semantics for a concurrent language.

1 Introduction

The purpose of this paper is to present the basic elements of a novel type of (non-standard) semantics for probabilistic concurrent languages, which is based on *linear spaces* and exploits functional analytical and operator algebraic notions and results. We will use this setting to shed additional light on the role synchronisation plays in modelling concurrent languages.

The semantics we propose can be seen as a denotational encoding of a transition system semantics, in as far as it represents transition graphs by linear operators (e.g. adjacency matrices) and at the same time allows for a compositional definition of these operators. In this way the usual distinction between operational vs denotational semantics becomes largely irrelevant. A similar weakening of this distinction can be found in a categorical context, e.g. in [61].

Quantitative structures such as linear spaces or linear algebras come naturally into consideration when giving the semantics of languages which contain quantitative elements such as probabilities.

We will define our semantics, which we call *linear semantics*, for a generic language including both probabilistic nondeterminism and synchronisation. This language represents a general scheme in which many probabilistic concurrent languages can be encoded, which belong to different programming paradigms (imperative, constraint, logic programming). The generic semantics is also a scheme and its concrete construction depends on the parameters provided by the particular language as well as on the specific properties of the language itself. The existence of such a semantics is nevertheless guaranteed in general and established by means of the Brouwer-Schauder Theorem in the abstract setting of C*algebras. The application of the general model to the cases of the constraint and imperative programming paradigms is exemplified in the appendix.

1.1 Plan of the Paper

In Section 2 we will introduce three toy languages which represent probabilistic extentions of well known declarative and imperative programming languages. Furthermore we will introduce a generic probabilistic language which subsumes these concrete toy languages.

Section 3 introduces the basic elements of our linear semantics by referring to concrete linear operators, i.e. finite and infinite dimensional matrices. The semantics of the previously discussed generic probabilistic language is given in Section 3.3 via a system of recursively defined equations. Furthermore, we discuss a notion of observables which captures the input/output behaviour of a program, and the instantiation of the generic semantics to the toy languages from Section 2.

In order to address the problem of whether there exist solutions to the semantical equations presented in Section 3 we will then, in Section 4, abstract the concrete (matrix) representation to an operator algebraic setting. By interpreting the semantical equations as equations over an abstract C*algebra we will then discuss various criteria which guarantee the existence of a solution.

Section 5 investigates some consequences and properties of the linear semantics. In particular we show how synchronisation-freeness of a language relates to the commutativity of the operators describing its semantics. This in turn leads to an interesting result about the structural complexity of the language.

In the concluding parts, Section 6 and Section 7, we discuss related work as well as some directions of further research. The Appendix assembles a number of small programs written in the toy languages introduced in Section 2, together with their linear semantics which illustrates our approach.

1.2 Quantitative Languages

A more general framework including probabilistic languages is the study of the semantics of *quantitative languages*. There are basically no restrictions on the syntactic form of a language which allows for a quantitative study instead of the usual qualitative one. We can speak of a quantitative language, if it syntactically distinguishes between "symbolic" (qualitative) terminals and numerical (quantitative) ones. The particular class of languages we are actually interested in will allow the quantification of *statements* and not just *expressions*.

Examples of quantitative languages are *probabilistic languages* where we assume as numerical domain $N = [0,1] \subset \mathbb{R}$. One could also think of *timed languages* with $N = \mathbb{R}$, or languages with $N = \mathbb{Z}$ which may refer to other quantities, e.g. the costs of some resources, etc.

1.3 Quantitative Semantics

The important issue when dealing with quantitative languages is that they require a semantics where it is possible to combine numbers, i.e. quantitative information, with semantical objects representing statements.

As a concrete example we would like to express the semantics of an agent A which is executed with probability p by means of a combination of the semantics of A and the semantics of p. Analogously, a quantitative semantics for a process P waiting for a time t should be defined by combining the semantics of t and the semantics of P. The canonical semantics of p and t is a number in some numerical domain $\mathcal{N}$.

As a consequence, we need to consider domains $\mathcal{D}$ which are the combination of a numerical domain $\mathcal{N}$ and a symbolic domain $\mathcal{S}$. In particular, we will consider *vector spaces* (over $\mathcal{N}$) as models [1] for the domains $\mathcal{D}$.

The choice of a vector space as a basic domain of our framework is particularly appropriate for our intended study of probabilistic languages. In fact, the standard vector space operations will allow us to express the most important constructs of a probabilistic language quite directly: probabilistic weighting of terms by the *scalar multiplication* "·", and the averaging among choices by the *vector addition* "+". As it will be clear later on, we will need to introduce some additional operations, thus extending the vector space structure into a more complex one. In particular, we will consider linear algebras and use their *vector multiplication* to model sequential composition [28].

[1] For some numerical domains, like $\mathbb{Z}$, it is more appropriate to consider structures like modules which although similar to vector spaces are based on rings instead of fields [32].

2 Probabilistic Languages

As an example of languages which incorporate some quantitative elements we will consider probabilistic languages. We introduce a generic language which allows us to express concurrency and probabilistic choice. The constructs of this language represent a scheme which can be instantiated to obtain concrete probabilistic languages. Programs in these languages have a natural operational interpretation as stochastic processes similar to the *random walk* on a graph, the latter being the graph associated to the transition system. Random walks are a kind of processes which are intensively studied in mathematics and several models have been introduced for them, mainly based on the theory of stochastic matrices and Markov chains [56,47,65]. It is therefore not surprising that we will look at similar structures, i.e. matrices and operators, for providing them with a denotational semantics.

As concrete examples, we will describe some toy languages belonging to different programming paradigms. We call them μ**PCCP**, μ**PCLP**, and μ**PImp**, where μ stands for "minimal", as we abstract from various "bureaucratic" operations, such as parameter passing, local variables, etc., and we reserve a very basic treatment to recursion.

2.1 Constraint Programming

Constraint Programming is a powerful paradigm which allows for a direct representation of a problem in terms of constraints [48]. A simple and elegant example of constraint programming paradigm is Concurrent Constraint Programming (CCP) [63]. The computational model of CCP is based on a global store represented by a constraint which expresses some partial information on the value of the variables involved in the computation. This replaces the imperative notion of store-as-evaluation. The concurrent execution of different processes refines the partial information of the store by adding new constraints via the primitive **tell**. In this way computations evolve monotonically. Communication and synchronisation are achieved by allowing processes to check via the primitive **ask** whether the store entails a constraint before proceeding with the computation. The model is based on the notion of a *constraint system* $\mathcal{C}$, defined as a complete algebraic lattice ordered with respect to an entailment relation $\vdash$; $\sqcup$ is the least upper bound (lub) operation, and *true*, *false* are the least and greatest elements of $\mathcal{C}$, respectively (see [63,17] for more details).

We discuss here two probabilistic constraint programming languages which are based on the CCP paradigm, namely μ**PCCP** and its synchronisation-free version μ**PCLP**.

2.1.1 μ**PCCP**

μ**PCCP** is a simplified version of Probabilistic Concurrent Constraint Programming (PCCP) which was introduced and discussed by the authors in

$$
\begin{array}{rcll}
A & ::= & \mathbf{tell}(c) & \text{add constraint to store} \\
 & & \Big| \; \square_{i=1}^{n} \, \mathbf{ask}(c_i) \rightarrow p_i : A_i & \text{probabilistic choice} \\
 & & \Big| \; \|_{i=i}^{n} \; p_i : A_i & \text{prioritised parallelism} \\
 & & \Big| \; proc & \text{procedure call}
\end{array}
$$

Fig. 1. The Syntax for Probabilistic Concurrent Constraint Programming.

[19,20,21,22]. It can be seen as a probabilistic version of Sarsawat et al.'s CCP language [63], although the latter is not subsumed by PCCP.

In Probabilistic Concurrent Constraint Programming randomness is expressed in the form of a *probabilistic choice*, which completely replaces the CCP nondeterministic choice and allows a program to make stochastic moves during its execution. We also replace the implicit non-deterministic scheduling in the interleaving semantics of the parallel construct by a probabilistic scheduler. This allows us to implement a kind of *prioritised parallelism*. By these constructs the element of chance is introduced directly at the algorithmic level without the need of any modification at the data level. Thus, the constraint system $\mathcal{C}$ underlying the language doesn't need to be re-structured according to some probabilistic or fuzzy or belief system (see e.g. [30,29] or [9]), and we can assume the definition of constraint system as a cylindric algebraic cpo given in [63], to which we refer for more details.

The syntax of μ**PCCP**-agents is given in Figure 1. The c and c_is are *finite* constraints in the constraint system $\mathcal{C}$ underlying the language. The p_is denote probabilities, representing a "weighting" on the sub-agents of the probabilistic choice and the parallel constructs.

A μ**PCCP** program P is a set of procedure declarations of the form $proc : -A$, with A a μ**PCCP**-agent. μ**PCCP** agents are essentially the same as CCP agents but for the probabilistic choice construct $\square_{i=1}^{n}\mathbf{ask}(c_i) \rightarrow p_i : A_i$ and the prioritised parallel construct $\|_{i=i}^{n} \; p_i : A_i$. We assume that the set $\{p_i\}_{i=1}^{n}$ forms a probability distribution both in the choice and in the parallel constructs, that is $\sum_{i=1}^{n} p_i = 1$.

The intuitive meaning of the probabilistic choice construct is as follows: First, check whether constraints c_i are entailed by the store current store d, i.e. $d \vdash c_j$. Then we have to *normalise* the probability distribution by considering only the enabled agents, i.e. the agents such that c_i is entailed and $p_i > 0$. This can be done by considering for enabled agents the normalised transition probability, $\tilde{p}_i = p_i / \sum_{d \vdash c_j} p_j$, where the sum is over all enabled agents. Finally, one of the enabled agents is chosen according this new probability distribution $\tilde{p}_i$.

We also replace the (implicit) non-determinism of the scheduler, which has

$$
\begin{array}{rcll}
A & ::= & \mathbf{tell}(c) & \text{add constraint to store} \\
 & & \Big|\ \Box_{i=1}^{n}\ p_i : A_i & \text{probabilistic choice} \\
 & & \Big|\ \|_{i=i}^{n}\ p_i : A_i & \text{prioritised parallelism} \\
 & & \Big|\ proc & \text{procedure call}
\end{array}
$$

Fig. 2. The Syntax for Probabilistic Constraint Logic Programming.

to decide in the interleaving semantics which agent has to be executed first, by a probabilistic version. This selection has to be made among all those agents A_i which are *active*, i.e. can make a transition. Then we have to normalise the priorities p_i of the active agents. This normalisation is done in the same way as described above for the probabilistic choice. Finally, the scheduler chooses one of the active agents according to the new probability distribution $\tilde{p}_i$. This allows us to keep track of all changes to the likelihood of a particular result during the entire computation.

2.1.2 μ**PCLP**

μ**PCLP** is a synchronisation-free version of μ**PCCP**. It is closely related to a probabilistic version of Constraint Logic Programming (CLP) [34], which justifies the name. The syntax is given in Figure 2 and is essentially the same as for μ**PCCP**, except that we eliminate guards. It can be seen as a variation of μ**PCCP** where all guards, i.e. the **ask** primitives, are (syntactically) restricted to be of the form **ask**(*true*), i.e. are always verified.

2.2 Imperative Programming

We describe in the following a simplified version of a probabilistic imperative language, μ**PImp**, based on the well-known **While** or **Imp** languages [54,70].

2.2.1 μ**PImp**

The syntax of μ**PImp** is shown in Figure 3. The intuitive semantics is the usual one for imperative languages, with the assignment as the basic state-changing operations. The state space has a flat structure and corresponds to the set, State = [Var $\mapsto$ Val], of all (partial) functions assigning some value in Val to the variables in Var. The semantics of the probabilistic choice is very similar to μ**PCCP**: First, check if the guards, i.e. $pred(x)$, hold, then choose a S_i according to normalised probabilities. Analogously, for the prioritised parallel construct we choose the S_i with higher probability after normalisation.

$$
\begin{array}{lcll}
S & ::= & x := v & \text{assignment} \\
 & & \mid\ \mathbf{begin}\ S_1\ ;\ S_2\ ;\ \ldots\ S_n\ \mathbf{end} & \text{sequential composition} \\
 & & \mid\ \mathbf{case}\ G_1\ \mathbf{or}\ G_2\ \mathbf{or}\ \ldots\ G_n\ \mathbf{esac} & \text{probabilistic choice} \\
 & & \mid\ \mathbf{par}\ W_1\ \mathbf{and}\ W_2\ \mathbf{and}\ \ldots\ W_n\ \mathbf{rap} & \text{prioritised parallelism} \\
 & & \mid\ proc & \text{procedure call} \\
G & ::= & \mathbf{if}\ pred(x)\ \mathbf{then}\ W & \text{guarded statement} \\
W & ::= & p : S & \text{weighted statement}
\end{array}
$$

Fig. 3. The Syntax of Probabilistic **Imp**

2.3 A Generic Language

In order to define our linear semantics, we introduce a generic language which reflects all the basic properties of probabilistic languages, and in particular those related to probabilistic nondeterminism and concurrency. In this way, we can define our model in all generality without committing with any particular language. This generic language subsumes all the concrete languages introduced above.

The syntax of the generic language is given in Figure 4. The b's represent *basic actions* (e.g. assignment, **tell**(c)). The g's stand for generic *guards*, i.e. tests for predicates (e.g. **ask**(c), $pred(x)$). The p's represent the quantitative information and in our case correspond to *probabilities*. Recursion is dealt with in its simplest form of argument-less procedures $proc$ which may be declared in an appropriate declaration section, e.g. $proc$ **is** A, or $proc : -A$, where A is an agent. We will denote by $\mathcal{A}$ the set of all agents in the generic language.

2.3.1 The Encoding of μ**PCCP** *and* μ**PCLP**

The syntax of μ**PCCP** and μ**PCLP**, as given in Figure 1 and in Figure 2, can easily be seen as an instance of the generic syntax in Figure 4.

We observe that μ**PCCP** is a restricted version, in the sense that we don't allow for an explicit sequential statement, although it is implicitly present in the interleaving semantics for the parallel construct [22]. The encoding is given by defining procedures as $proc : -A$, and

- Basic Action, b: **tell**(c), and
- Basic Guard, g: **ask**(c).

The translation for μ**PCLP** is the same as for μ**PCCP**, besides that guards are ignored or occur in the form **ask**$(true)$.

$$
\begin{array}{rcll}
A & ::= & b & \text{basic action} \\
 & | & p : A & \text{weighted statement} \\
 & | & g \rightarrow A & \text{guarded statement} \\
 & | & \bigodot_{i=1}^{n} A_i & \text{sequential composition} \\
 & | & \square_{i=1}^{n} A_i & \text{choice} \\
 & | & \|_{i=1}^{n} A_i & \text{parallelism} \\
 & | & proc & \text{procedure call}
\end{array}
$$

Fig. 4. The Syntax of the Generic Language

2.3.2 The Encoding of μ**PImp**

The syntax in Figure 3 differs slightly more from the one in Figure 4 than in the case of μ**PCCP** and μ**PCLP**. Its translation is however quite obvious. The relevant point is how to define basic actions and guards:

- Basic Action, b: $x := v$, and
- Basic Guard, g: $pred(x)$.

3 Linear Semantics

In this section we will introduce a concrete representation of a linear semantics by means of linear operators defined on a suitable linear space and representing each construct of the generic language in Figure 4. The basic idea of the linear semantics is to lift the “state space” of the underlying deterministic language to a vector space.

In order to define a denotational semantics we then look at an appropriate space of “state-transforming” mappings reflecting the computational evolution during the program's execution. These mappings will be defined as linear operators on the before mentioned linearised “state space”. The operators form a linear algebra in the sense of [28], which is the reason why we refer to the corresponding semantics as “linear semantics”.

3.1 Lifting the State Space

The particular construction we will utilise is the free vector space over the basic state space, State, of the underlying deterministic language. This is a constraint set $\mathcal{C}$ for constraint languages, or a function space [Var $\mapsto$ Val] for imperative languages, and contain all the necessary information describing a certain situation during a computation. We assume that State is a finite or countable infinite set. The free vector space $\mathcal{V}$(State) is constructed as the set of all formal linear combinations of states:

$$\mathcal{V}(\text{State}) = \left\{ \sum_{s \in \text{State}} x_s s \mid x_s \in \mathbb{R},\ s \in \text{State} \right\}.$$

This construction consists in taking the elements in State as the generators of the vector space, that is as its base vectors. Note that depending on the cardinality of State we may have finite or countable infinite linear combinations. Free vector spaces stem from a purely algebraic construction, and no particular topological structure (with associated convergence notion) is assumed.

For finite state sets, State, the free vector space is isomorphic to the standard real vector space $\mathbb{R}^n$, where n is the cardinality of State.

We can denote a vector in $\mathcal{V}(\text{State})$ equivalently either as a formal sum

$$\vec{x} = \sum_{s \in \text{State}} x_s \vec{s},$$

or as an n-tuple

$$\vec{x} = (x_s)_{s \in \text{State}},$$

or as a set of pairs

$$\vec{x} = \{\langle s, x_s \rangle\}_{s \in \text{State}}.$$

This construction can be seen as a quantitative analogue of the usual powerset construction $\mathcal{P}(\text{State})$ often used in the denotational semantics of non-deterministic languages [57,70], and is alternative to the probabilistic power domain construction introduced in [37,36].

Example 3.1 Assume State $= \mathbb{N}$. Then $\mathcal{V}(\text{State}) = \mathcal{V}(\mathbb{N})$ is the infinite dimensional vector space $\mathbb{R}^\omega$ with typical element

$$\vec{x} = \sum_{i \in \mathbb{N}} x_i \vec{i},$$

where $x_i \in \mathbb{R}$ and $\vec{i}$ represents the base vector corresponding to the natural number i. Each set in the powerset of $\mathbb{N}$ can be represented by a vector in $\mathcal{V}(\mathbb{N})$. For example the set $\{1, 3, 27\}$ corresponds to the vector $\vec{1} + \vec{3} + \vec{27}$.

3.2 Graphs and Algebras

A semantics for a probabilistic language which describes the "effects" of every program in this language on the computational states can be defined by assigning to each agent or statement in the language a linear operator — i.e. a possibly infinite matrix — on the free vector space $\mathcal{V}(\text{State})$ over the underlying "state space" which encode all the possible computational paths (traces). Each entry $(\mathbf{M})_{cd}$, $c, d \in \text{State}$, of such an operator $\mathbf{M}$ encodes the possibility or probability to reach a state d starting from another state c.

For deterministic and purely non-deterministic languages such entries can be defined as follows:

$$(\mathbf{M})_{cd} = \begin{cases} 1 \text{ iff } c \longrightarrow^* d \\ 0 \quad \text{otherwise} \end{cases}$$

where $c \longrightarrow^* d$ means that there is a computation leading from state c to state d. In the deterministic case, where there is at most one transition from each state, we see that we get at most one entry in each row of $\mathbf{M}$. For non-deterministic languages, we may have several non-zero entries [2].

In our case of probabilistic languages, the entries in $\mathbf{M}$ correspond to the probabilities associated to the computational path leading from state c to state d:

$$(\mathbf{M})_{cd} = \begin{cases} p \text{ iff } c \longrightarrow_p^* d \\ 0 \quad \text{otherwise} \end{cases}$$

$\mathbf{M}$ is then a so-called *stochastic matrix*, i.e. a matrix where the sum of all entries in each row is one.

For (non-)deterministic languages the operator $\mathbf{M}$ corresponds to a generalisation of the well-known adjacency matrix [8,50]. For probabilistic languages we can also see these operators as transformations of distributions which describe the current state of a computation into another. These matrices therefore are generators of a well known class of stochastic processes, namely Markov chains [65,67].

A reasonable question to ask is whether this “encoding of transitions” really can result in a denotational semantics and not “just” in another operational semantics.

There are several answers to this issue: First, the semantics we will give is indeed defined compositionally via a set of recursive semantical equations. The reason why this is possible is that the “constructors” we use to define the semantics of an agent in terms of its constituents are not “graph-based” but expressed in terms of the corresponding operators. For example, the product of two adjacency matrices is not directly translatable into operations on the original graphs, while it is always possible to multiply two operators. Second, we believe that the distinction between operational semantics and denotational semantics is somehow arbitrary: If the operational semantics of the constituents of a composite agent indeed encode all semantic information, then it must be possible to construct the semantics of the composite agent given this information and the structure of the composition, although this construction may be not expressible in a straightforward way.

[2] For finitely-branching languages as the ones we consider, we will have only finitely many non-zero entries in each row (row-finiteness).

$[\![b]\!]$	$=$	$\mathbf{B}$	basic action
$[\![p : A]\!]$	$=$	$p \cdot [\![A]\!]$	weighted statement
$[\![g \rightarrow A]\!]$	$=$	$\mathbf{G} \cdot [\![A]\!]$	guarded statement
$[\![\bigodot_{i=1}^{n} A_i]\!]$	$=$	$\prod_{i=1}^{n} [\![A_i]\!]$	sequential composition
$[\![\square_{i=1}^{n} A_i]\!]$	$=$	$\sum_{i=1}^{n} [\![A_i]\!]$	choice
$[\![\Vert_{i=1}^{n} A_i]\!]$	$=$	$\bigoplus_{\pi \in \mathcal{P}(\{1,\ldots,n\})} \left(\prod_{i=1}^{n} [\![A_{\pi(i)}]\!]\right)$	parallelism
$[\![proc]\!]$	$=$	$[\![A]\!]$	procedure call

Fig. 5. A Generic Semantics

3.3 A Generic Semantics

In this section we present a core semantics which allows for a uniform treatment of the probabilistic languages introduced in Section 2. Based on the intuitions about the "transitional" nature of linear operators and matrices we will formulate a system of equations which compositionally describe the semantics of an agent and in particular of recursive procedure calls.

We associate to each agent $A \in \mathcal{A}$ in the generic language a linear operator $[\![A]\!] : \mathcal{V}(\text{State}) \mapsto \mathcal{V}(\text{State})$. We call the map $[\![.]\!]$ from the set of agents $\mathcal{A}$ into the algebra, $\mathcal{L}(\mathcal{V}(\text{State}))$, of linear operators on the free vector space $\mathcal{V}(\text{State})$ an *interpretation* of agents. The semantics of an agent in the generic language is then defined recursively via the system of semantical equations over the set $\mathcal{I}$ of all interpretations defined in Figure 5.

The semantics of the various agents in the generic language is specified by means of the basic operations of multiplication (to model sequential composition and guarded statement), sum (to model choice) and scalar multiplication (to model weighted statements). We will use the same symbol "$\cdot$" for both the operator multiplication and the scalar multiplication. Moreover, we use the *direct sum* operation to model the parallel construct as interleaving. Given a set of operators $\mathbf{M}_i$ their direct sum is defined as follows:

$$\bigoplus_i \mathbf{M}_i = \begin{pmatrix} \mathbf{M}_1 & 0 & 0 & \ldots \\ 0 & \mathbf{M}_2 & 0 & \ldots \\ 0 & 0 & \mathbf{M}_3 & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{pmatrix},$$

where the $\mathbf{M}_i$ are referred to as the *factors* of $\bigoplus \mathbf{M}_i$. Each factor represents a different interleaving. Thus, we have to consider as many copies $\mathcal{V}(\text{State})$ as the number of possible interleavings. This is equivalent to combining several copies of the algebra $\mathcal{L}(\mathcal{V}(\text{State}))$ of linear operators on the vector space [3]. Thus, interpretations will actually be maps from $\mathcal{A}$ into a direct sum of copies of $\mathcal{L}(\mathcal{V}(\text{State}))$.

A semantics will be defined whenever there exists a solution for these equations. A standard way to construct such a solution is to associate a program (i.e. a set of procedure declarations) with a mapping on interpretations and then look for a fixpoint of this mapping. For nondeterministic languages, this is typically defined as a monotonic function on a complete lattice, for which the Knaster-Tarski theorem guarantees the existence of a maximum and minimum fixpoint [70].

For probabilistic languages the existence of a fixpoint can be shown in a non-constructive way by means of the Brouwer Fixpoint theorem and various of its generalisation (cf. [21]). We will discuss this point further in Section 4.

3.3.1 Interpretations

The equations and operations used in Figure 5 are to be read as follows.

The first equation simply states that there is an operator $\mathbf{B}$ which represents a basic action. As we did not specify the nature of the set State and the effect of b on it, it will be left open for the concrete instantiation of the semantics to specify $\mathbf{B}$.

The second equation "weights" each operator according to the probability p. This weighting is important in the context of probabilistic choices and prioritised interleaving.

The third equation introduces a guard operator $\mathbf{G}$, which is again language dependent. In principle, we will make no further restrictions on $\mathbf{G}$ except that it is a *projection operator*, that is $\mathbf{G}^2 = \mathbf{G}\mathbf{G} = \mathbf{G}$. This is because we will expect a "double guarded" agent to behave in the same way as a "single guarded" one, i.e.

$$[\![g \to (g \to A)]\!] = [\![g \to A]\!].$$

The forth equation defines the sequential composition of of n agents $\{A_i\}_{i=1}^n$ by the functional composition of the A_i's represented by the matrix (or, more in general, operator) multiplication.

The fifth equation defines the semantics of the choice among n agents as the sum of the corresponding operators. When the agents of the choice are guarded weighted statements (probabilistic choice) then the semantical operator corresponds to a weighted sum of the operators associated to each sub-agent. This can also be seen as averaging the effects of sub-agents according to their probability.

The sixth equation deals with parallelism. We assume here that the actions are atomic, that is non-interruptable. This makes the model simpler as we can express all possible interleaving sequences as permutations rather than

[3] Direct sums can be seen as a way to represent or simulate sets or distributions over operators within the framework of operator algebras.

via more complicated combinatorial formulas. The direct sum allows us to keep each possible interleaving in a separate (possible) world represented by a factor in the sum. Such a factor is the sequential composition of the sub-agents A_i according to the corresponding permutation π. In Figure 4 $\mathcal{P}(\{1,\ldots,n\})$ denotes the set of all permutations of n elements.

The last equation expresses the semantics of a procedure call. As already mentioned we abstract from the details of parameter passing and local variables. So the meaning of a procedure call is simply the meaning of the agent defining that procedure in the program.

3.3.2 Observables

The equations in Figure 5 establish the interpretation of all the constructs in the language, but do not identify a unique semantics $[\![\cdot]\!]$. This is essentially due to recursion. In fact, for a procedure declaration of the form $proc : -proc$, every linear operator $[\![proc]\!]$ satisfies the equations. The choice of a particular interpretation depends on the notion of observables we want to model. For example, if we want to observe finite behaviours then $[\![proc]\!]$ should be the null operator because there are no finite computations for $proc$. On the other hand, if we are interested in observing infinite computations then $[\![proc]\!]$ should be the identity operator since any computation starting with $proc$ will not affect any state; in fact, it will not be sensed by an external observer who can only look at the evolution of the store.

A classical notion of observables is the input/output behaviour of an agent A, that is the results (outputs) of both finite and infinite computations starting from a given initial store (input). Such observables, which we will denote by $\mathcal{O}(A)$, can be retrieved by the linear operator $\mathbf{M} = [\![A]\!]$ associated to the agent A by means of some appropriate transformations which we will describe in the following.

Each row of $\mathbf{M}$ describes the possible transitions of the corresponding state together with their probability. For some initial states an agent may be "inactive" or "stuck", i.e. no transitions are possible. This situation would correspond in the matrix $\mathbf{M}$ for that agent to a row containing only zero entries. If we look at $\mathbf{M}$ as a state transformer, we can easily see that these zero-rows may "destroy" the store instead of leaving it unchanged, as one would intuitively expect from an inactive agent. In order to avoid this problem we introduce the following operation:

$$(I(\mathbf{M}))_{cd} = \begin{cases} 1 & \text{for } c = d \text{ and } \forall d : (\mathbf{M})_{cd} = 0 \\ (\mathbf{M})_{cd} & \text{otherwise,} \end{cases}$$

whose effect is to put a 1 in the diagonal elements of zero-rows.

Another question is related to normalisation. Sometimes, the probabilities in each row must be re-normalised. This is necessary when some of the guarded agents in a choice are not enabled or when transitions from the same state occur in more than one interleaving. For the j-th factor in a direct sum $\bigoplus_i \mathbf{M}_i$, the *normalisation* $N(\mathbf{M}_j)$ is defined as:

$$(N(\mathbf{M}_j))_{cd} = \begin{cases} 0 & \text{iff } \sum_i \sum_d (\mathbf{M}_i)_{cd} = 0 \\ \frac{(\mathbf{M}_j)_{cd}}{\sum_i \sum_d (\mathbf{M}_i)_{cd}} & \text{otherwise} \end{cases}$$

Finally, transitions may occur in different interleavings, which have the same result. We then need to identify all such transitions by summing up all the associated probabilities. This *compactification* can be achieved by summing all the factors in a direct sum:

$$K\left(\bigoplus_i \mathbf{M}_i\right) = \sum_i \mathbf{M}_i.$$

With these operations we can retrieve the input/output observables $O(A)$ of an agent A from its linear semantics $[\![A]\!]$ by defining:

$$\mathcal{O}(A) = K(N(I([\![A]\!]))).$$

3.4 Linear Algebra vs Functional Analysis

In the case in which the set State is finite the linear semantics can be given within the framework of finite dimensional vector spaces, i.e. in terms of *linear algebra*. In this case we can represent all operators as finite matrices, which simplifies our constructions substantially.

The situation gets more complicated once we want to consider (countable) infinite state spaces. These are essentially unavoidable as soon as recursion is present in the language. Unbound recursion also poses of dealing with infinite processes and their limits. In other words, we have to consider not only the algebraic structure of the free vector space $\mathcal{V}(\text{State})$ and the space $\mathcal{L}(\mathcal{V}(\text{State}))$ of linear operators on it, but additionally we need some kind of *topology* on them.

In *functional analysis* — i.e. in dealing with infinite dimensional vector spaces — a common way to introduce a topology on a linear space is by means of a *norm*, $\|\cdot\|$ and its corresponding metric $d(\vec{x},\vec{y}) = \|\vec{x},\vec{y}\|)$. The topology induced by such a metric is called *norm topology* [25,71,40].

A nice feature of finite dimensional linear spaces is the fact that all norm topologies are equivalent [26]. Therefore, one can refer to *the* topology of $\mathcal{V}(\text{State})$ and $\mathcal{L}(\mathcal{V}(\text{State}))$ without specifying any particular norm.

In the infinite case, different norms give rise to different topologies and it is therefore impossible to consider a "standard" norm or topology on those spaces. Infinite dimensional spaces like $\mathcal{V}(\text{State})$ and $\mathcal{L}(\mathcal{V}(\text{State}))$ are thus less amenable to a semantical treatment than finite dimensional ones since the semantics depends on the particular topology.

For the infinite dimensional case we will consider here a topology on $\mathcal{V}(\text{State})$ which stems from a particular norm, namely the 2-norm, often referred to as the standard *Euclidean* distance, which is defined as

$$\|\vec{x}\|_2 = \|(x_c)_{c\in\text{State}}\|_2 = \sqrt{\sum_{c\in\text{State}} |x_c|^2},$$

This means in effect that instead of considering the purely algebraic free vector space $\mathcal{V}(\text{State})$ we will base our constructions on the *Hilbert space* $\ell^2(\text{State})$ [59,40]:

$$\ell^2(\text{State}) = \{\vec{x} \in \mathcal{V}(\text{State}) \mid \|\vec{x}\|_2 < \infty\}$$

We also have to introduce a topology on the space of liner operators $\mathcal{L}(\mathcal{V}(\text{State}))$ or more precisely on the space $\mathcal{L}(\ell^2(\text{State}))$. In line with common functional analytical practice we will consider as domain only continuous, linear operators on $\ell^2(\text{State})$. Since for linear operators continuity and boundedness are equivalent, it is common to denoted them as: $\mathcal{B}(\ell^2(\text{State}))$ [59,40].

3.5 Instantiating the Generic Semantics

For each language presented in Section 2 we can instantiate our framework by providing a concrete state space (i.e. the basis for the appropriate free vector space construction), and by specifying the concrete operators for the basic agents as well as the guards.

Essential for our construction is the use of a special "one-step" operator. The unit matrix $\mathbf{E}_{cd}$ relating two states c and d in State is defined as:

$$\mathbf{E}_{cd} = \begin{cases} 1 \text{ at entry } c, d \\ 0 \text{ otherwise} \end{cases}$$

$\mathbf{E}_{cd}$ is a matrix containing only one non-zero entry which represents exactly the transition from state c to d. It is obvious that $\mathbf{E}_{cd}$ is a linear operator, i.e. an element of $\mathcal{L}(\mathcal{V}(\text{State}))$. Note that in the finite dimensional case the set of all operators of this type, for all states c and d, form an algebraic basis of the space of linear operators on the vector space $\mathcal{V}(\text{State})$.

*3.5.1 Constraint Programming: μ**PCLP** and μ**PCCP***

The underlying state set is given by a (cylindric [4]) constraint system $\mathcal{C}$, which is basically a cpo with bottom [17]. The linearisation of this set leads to the vector space $\mathcal{V}(\mathcal{C})$ defined by

$$\mathcal{V}(\mathcal{C}) = \left\{ \sum_{c\in\mathcal{C}} x_c c \mid x_c \in \mathbb{R},\ c \in \mathcal{C} \right\}.$$

The basic action $\mathbf{tell}(c)$ adds a constraint to the current store. This is a deterministic operation, thus it can be modelled by a 0/1 matrix. The meaning of agent $\mathbf{tell}(c)$ is defined so as to encode its effect on all possible

[4] We don't consider hiding and local variables here, therefore we can ignore the cylindric nature of $\mathcal{C}$.

stores. When the initial store is d and we execute **tell**(c) the resulting store is given by the least upper bound, $d \sqcup c$, of c and d. This can be represented by the matrix:

$$[\![\mathbf{tell}(c)]\!] = \sum_{d \in \mathcal{C}} \mathbf{E}_{d,d \sqcup c}.$$

One can easily see that all these matrices encode the least upper bound operation of the underlying constraint system completely: For any given c the matrix $[\![\mathbf{tell}(c)]\!]$ represents exactly a row in the table for the "$\sqcup$" operation.

Proposition 3.2 *The operators representing μ***PCLP** *agents commute.*

Proof. Since the least upper bound operator $\sqcup$ is commutative, we have

$$[\![\mathbf{tell}(c) \odot \mathbf{tell}(d)]\!] = [\![\mathbf{tell}(c \sqcup d)]\!] = [\![\mathbf{tell}(d) \odot \mathbf{tell}(c)]\!].$$

Therefore,

$$[\![\mathbf{tell}(c)]\!] \cdot [\![\mathbf{tell}(d)]\!] = [\![\mathbf{tell}(d)]\!] \cdot [\![\mathbf{tell}(c)]\!]$$

for all constraints c and d in $\mathcal{V}(\mathcal{C})$. **tell** agents are the only basic agents for μ**PCLP**. The guards are removed or occur in the form **ask**(true). The latter corresponds to the identity which commutes with all operators.

Furthermore we observe that the operations used in the semantical equations in Figure 5 are commutative, e.g. $\mathbf{A}+\mathbf{B} = \mathbf{B}+\mathbf{A}$. By a simple structural induction we can then show that if $[\![A_i]\!]$ commute then so does their linear combination, weighted product and direct sum. $\square$

For monotonic enumerations of the constraint system $\mathcal{C}$ (i.e. stronger constraints get larger indices), the operators representing any agents correspond to upper-triangular matrices.

Proposition 3.3 *The operators representing μ***PCLP** *agents* **tell** *are upper-triangular.*

Proof. By structural induction, the fact that computations in μ**PCLP** are monotone, and the closure of upper-triangular matrices with respect to the μ**PCLP** operators. $\square$

The language μ**PCCP**provides also a construct for synchronisation, that is the **ask** operator. As discussed in the general case, this operator can be modelled by a projection. One way to represent the guard **ask**(c) in **ask**$(c) \rightarrow p : A$ is via a diagonal matrix which implements a projection onto the subspace of stores which entail c.

$$[\![\mathbf{ask}(c)]\!] = \sum_{c \sqsubseteq d} E_{dd},$$

where E_{dd} is the matrix with only one non-zero entry which is (d, d).

Each guarded agent **ask**$(c) \rightarrow p : A$ in a choice construct can be then modelled by multiplying this projection operation with the operator $[\![A]\!]$ representing the agent A. This ensures that only those transitions described in $[\![A]\!]$ actually take place, which start from a store entailing c.

Note that the matrices $[\![\mathbf{ask}(c)]\!]$ encode the partial order "$\sqsubseteq$" in the constraint system: For every c, the projection operator $[\![\mathbf{ask}(c)]\!]$ represents the upward closure of c.

Proposition 3.4 *The operators for μ***PCCP** *agents are upper-triangular.*

Proof. Assume a monotonic enumeration of the constraint system. Then since computations in μ**PCCP** are monotonic and projections are obviously upper-triangular, the basic agents are upper-triangular. The inductive cases follows by the closure of upper-triangular matrices with respect to the μ**PCCP** operators. □

The following example show that the linear operators representing μ**PCCP** agents in general do not commute.

Example 3.5 Consider the two agents:

$$A \equiv \mathbf{ask}(c) \to \frac{1}{2} : \mathbf{tell}(d) \,[]\, \mathbf{ask}(\mathit{true}) \to \frac{1}{2} : \mathbf{tell}(e) \quad \text{and} \quad B \equiv \mathbf{tell}(c)$$

The semantics of A and B do not commute:

$$[\![A]\!] \cdot [\![B]\!] = [\![\mathbf{tell}(e)]\!][\![\mathbf{tell}(c)]\!] = [\![\mathbf{tell}(c \sqcup e)]\!]$$
$$[\![B]\!] \cdot [\![A]\!] = \frac{1}{2}\left([\![\mathbf{tell}(c \sqcup e)]\!] + [\![\mathbf{tell}(c \sqcup d)]\!]\right).$$

3.5.2 Imperative Programming: μ**PImp**

The linearisation of the set State $=$ [Var $\mapsto$ Val], leads to the vector space $\mathcal{V}([\mathrm{Var} \mapsto \mathrm{Val}])$. For the sake of simplicity, we consider here only assignments involving constants, and represent a state as a sequence:

$$(x_1 = v_1, x_2 = v_2, \ldots) \equiv (x_i = v_i)_i$$

indicating that a certain state is represented by the fact that each variable x_i has value v_i. We can introduce a symbol $\perp$ to indicate undefined variables.

Since we allow only deterministic assignments [5], the basic actions are again 0/1 matrices. The operator for $x := v$ encodes those transitions between states which are identical except that the value x is bound to is changed:

$$[\![x := v]\!] = \sum_{(\ldots,x=w,\ldots)\in\mathrm{State}} \mathbf{E}_{(\ldots,x=w,\ldots),(\ldots,x=v,\ldots)}.$$

In general, two assignments to the same variable do not commute. Therefore, we cannot expect the corresponding operators to commute.

Example 3.6 Consider, the following two agents:

$$A \equiv x := 3 \quad \text{and} \quad B \equiv x := 4.$$

[5] We introduce probabilities at the algorithmic level, in the parallel and choice constructs. Other approaches are based on a "random assignment" and use general stochastic matrices [44,30].

$\Phi(\llbracket b \rrbracket)$	$=$	$\mathbf{B}$	basic action
$\Phi(\llbracket p : A \rrbracket)$	$=$	$p \cdot \llbracket A \rrbracket$	weighted statement
$\Phi(\llbracket g \rightarrow A \rrbracket)$	$=$	$\mathbf{G} \cdot \llbracket A \rrbracket$	guarded statement
$\Phi(\llbracket \bigodot_{i=1}^{n} A_i \rrbracket)$	$=$	$\prod_{i=1}^{n} \llbracket A_i \rrbracket$	sequential composition
$\Phi(\llbracket \, []_{i=1}^{n} A_i \rrbracket)$	$=$	$\sum_{i=1}^{n} \llbracket A_i \rrbracket$	choice
$\Phi(\llbracket \, \|_{i=1}^{n} A_i \rrbracket)$	$=$	$\bigoplus_{\pi \in \mathcal{P}(\{1,\ldots,n\})} \left(\prod_{i=1}^{n} \llbracket A_{\pi(i)} \rrbracket\right)$	parallelism
$\Phi(\llbracket proc \rrbracket)$	$=$	$\llbracket A \rrbracket$	procedure call

Fig. 6. A Generic Fixpoint Operator

The semantics of their sequential composition is:

$$\llbracket A \rrbracket \cdot \llbracket B \rrbracket = \llbracket B \rrbracket \quad \text{while} \quad \llbracket B \rrbracket \cdot \llbracket A \rrbracket = \llbracket A \rrbracket.$$

In general, operators representing assignments are neither commutative nor of any special form. For example, they are not upper-triangular because the computation is, contrary to the constraint programming case, not necessarily monotonic.

Guards can be treated like in the constraint case by projections. We just have to construct a diagonal operator which selects those states which fulfil a certain condition $pred(x)$:

$$\llbracket pred(s) \rrbracket = \sum_{\{s \in \text{State}|pred(s)\}} \mathbf{E}_{ss}.$$

4 Abstract Semantics

The linear semantics discussed in Section 3 is based on a domain of linear operators which encode the operational semantics of a program by recording the steps of the program in the transition graph.

The fact that we consider languages with possibly unbounded recursion makes it impossible to give a bound for the dimension of the algebra of linear operators. Thus, our semantical domain cannot be in general defined as a finite dimensional algebra, $\mathcal{L}(\mathbb{R}^n)$ for some $n < \infty$. On the other hand, the choice of algebras of linear operators on infinite dimensional vector spaces introduces the problem of finding the appropriate topology for dealing with recursion. This is important in order to find a solution to the equations in Figure 5.

In this section we will address this problem from a general viewpoint by singling out the conditions which guarantee the existence of a solution. To this purpose, we will consider the abstract setting of C*algebras which is general enough to subsume both finite dimensional vector spaces and various infinite dimensional operators structures. Moreover, C*algebras provide a natural interpretation of all the needed operations and are topologically complete.

4.1 C Algebras*

C*algebras (cf. [23], [40,41], [53], [15], [24]) represent a combination of algebraic and topological structures, which can be seen as continuous, infinite dimensional generalisations of matrix algebras. The theory of C*algebras was pioneered in the 1940s by Gelfand and Naimark [24] and play an important role in many areas not least in quantum physics [66].

We recall here the basic definitions:

Definition 4.1 A linear algebra $\mathcal{A}$ is a complex vector space on which there is a mapping $\cdot : \mathcal{A} \times \mathcal{A} \mapsto \mathcal{A}$ called *multiplication*, having the properties (for all $a, b, b_i, c \in \mathcal{A}$ and $\alpha \in \mathbb{C}$):

$$a(b_1 + b_2) = ab_1 + ab_2, \quad a(bc) = (ab)c, \quad a(\alpha b) = \alpha ab.$$

If there exists a *unit* element $\mathbf{Id}$ such that $\mathbf{Id} \cdot a = a \cdot \mathbf{Id}$ for all $a \in \mathcal{A}$ we speak of a *unital algebra*.

A $*$-*algebra* $\mathcal{A}$ is an algebra on which there is a mapping $.^* : \mathcal{A} \mapsto \mathcal{A}$ called *conjugation* or *adjoint*, having the properties (for all $a, b \in \mathcal{A}$ and $\alpha \in \mathbb{C}$):

$$(ab)^* = b^* a^*, \quad (a + b)^* = a^* + b^*, \quad (\alpha a)^* = \alpha^* a, \quad a^{**} = a.$$

The last property is called *involution*. A *C* algebra* $\mathcal{A}$ is a $*$-algebra and a Banach space, the norm of which satisfies (for all $a, b \in \mathcal{A}$ and $\alpha \in \mathbb{C}$):

$$\|ab\| \leq \|a\|\|b\|, \quad \|a^*\| = \|a\|, \quad \|aa^*\| = \|a\|\|a^*\|, \quad \|\mathbf{Id}\| = 1.$$

In short: A C*algebra is an involutive Banach algebra $\mathcal{A}$ with $\|a^* a\| = \|a\|^2$, for all $a \in \mathcal{A}$.

Various important structures like the algebra of continuous functions and matrix algebras are examples of C*algebras. In order to give a rough idea of what C*algebras are we mention four prototypical examples:

Example 4.2 The simplest example of a C*algebra is given by the set of complex numbers $\mathbb{C}$ with the usual algebraic operations (note that here scalar and algebra product actually coincide). The involution corresponds to complex conjugation $\overline{x + iy} = x - iy$, and the norm is the usual absolute value $\|x + iy\| = |x + iy| = \sqrt{x^2 + y^2}$.

Example 4.3 The set of all continuous complex-valued functions on a compact Hausdorff space $C(X, \mathbb{C})$ forms a C*algebra. Let $f, g \in C(X, \mathbb{C}), \alpha \in \mathbb{C}$, then the C*algebra operations are defined point-wise (for $x \in X$) as follows: $(\alpha f)(x) = \alpha f(x)$, $(f+g)(x) = f(x)+g(x)$, $(fg)(x) = f(x)g(x)$, and $(f^*)(x) = \overline{f(x)}$. The C*norm is the supremum norm: $\|f\| = \|f\|_\infty = \sup_{x \in X} f(x)$.

Example 4.4 Another basic example of a C*algebra is given by the $n \times n$ matrices over $\mathbb{C}$. The algebraic operations are defined in the usual way. Let $\mathbf{S} = (s_{ij})_{i,j=1}^n$ and $\mathbf{T} = (t_{ij})_{i,j=1}^n$ be two $n \times n$ matrices over $\mathbb{C}$. The scalar product is defined by $\alpha\mathbf{T} = \alpha(t_{ij})_{i,j=1}^{n} = (\alpha t_{ij})_{i,j=1}^{n}$, the addition by $\mathbf{T} + \mathbf{S} = (t_{ij})_{i,j=1}^{n} + (s_{ij})_{i,j=1}^{n} = (t_{ij} + s_{ij})_{i,j=1}^{n}$ and the algebra product by $\mathbf{TS} = (t_{ij})_{i,j=1}^{n}(s_{ij})_{i,j=1}^{n} = (\sum_{k=1}^{n} t_{ik}s_{kj})_{i,j=1}^{n}$. The involution is defined as complex conjugation of the transposed matrix, i.e. $\mathbf{T}^* = ((t_{ij})_{i,j=1}^{n})^* = (\overline{t_{ji}})_{i,j=1}^{n}$. The appropriate C*norm for $\mathbb{M}_n(\mathbb{C})$ is the so called operator norm (with $x \in \mathbb{C}^n$): $\|T\| = \sup_{\|x\|=1} \|T(x)\|$.

Example 4.5 Let $\mathcal{H}$ be a Hilbert space and let $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{H}, \mathcal{H})$ be the set of linear operators on $\mathcal{H}$. We can generalise the operator norm from the above example to this infinite dimensional spaces by defining: $\|\mathbf{T}\| = \sup_{\|x\|=1} \|\mathbf{T}(x)\|$, where $x \in \mathcal{H}$ and $\mathbf{T} \in \mathcal{L}(\mathcal{H})$). The set of all bounded linear operators on $\mathcal{H}$, i.e. operators for which $\|\mathbf{T}\| < \infty$ holds, is denoted by $\mathcal{B}(\mathcal{H})$. It forms a C*algebra with respect to the above operator norm and the common vector space operations: scalar product $(\alpha\mathbf{T})(x) = \alpha\mathbf{T}(x) = \mathbf{T}(\alpha x)$, addition $(\mathbf{T} + \mathbf{S})(x) = \mathbf{T}(x) + \mathbf{S}(x)$ and algebra product: $(\mathbf{TS})(x) = \mathbf{T}(\mathbf{S}(x))$; finally the (unique) involution on $\mathcal{B}(\mathcal{H})$ is defined by the condition: $\langle \mathbf{T}(x), y\rangle = \langle x, \mathbf{T}^*(y)\rangle$

Examples 4.4 and 4.5 show that matrices and infinite operators are prototypical examples of C*algebras. Thus, the constructions in Section 3 fits into a C*algebraic framework, as they are based on finite dimensional spaces or infinite dimensional Hilbert spaces, and a concrete semantics is obtained by solving the semantical equations in Figure 5 in a concrete C*Algebra, namely $\mathcal{B}(\mathcal{V}(\text{State}))$ (see Example 4.5). Therefore, we can safely generalise our reasoning to solving such equations in an abstract C*algebra. The Gelfand-Naimark Theorem formalises the relationship between the two settings.

Definition 4.6 A *representation* of a C*algebra $\mathcal{A}$ on a Hilbert $\mathcal{H}$ is a *-homomorphism, π, from $\mathcal{A}$ into $\mathcal{B}(\mathcal{H})$. It is said to be *faithful* if π is actually a *-isomorphism.

The Gelfand-Naimark Theorem guarantees that each C*algebra has a faithful representation on some Hilbert space.

Theorem 4.7 *(Gelfand-Naimark) Given a C*-algebra $\mathcal{A}$ then there exists a (faithful) representation of $\mathcal{A}$ as bounded linear operators on a Hilbert space.*

In Example 4.3 the algebra of continuous (complex) functions is identified with a C*algebra. The next theorem shows that they actually form the prototypical example of an *Abelian* C*algebra, that is a C*algebra where the multiplication is commutative. The Gelfand Duality Theorem (see e.g. [40, Thm 4.4.1], [53, Thm 2.1.10], [15, Thm I.3.1]) establishes a universal representation for Abelian C*algebra.

Theorem 4.8 *(Gelfand Duality) Suppose $\mathcal{A}$ is an Abelian C*algebra, then it is isometric, *-isomorph to the algebra of continuous functions $C_0(\mathcal{M})$ on a (locally) compact Hausdorff space $\mathcal{M}$.*

4.2 Existence of a Semantics

As already mentioned in Section 3.3, the problem of defining a semantics for our generic language can be addressed as the problem of guaranteeing a solution to the recursive equations in Figure 5. The usual approach is to reformulate the problem as a fixpoint problem, i.e. to encode the semantical equations in the definition of an operator whose fixpoints are solutions of the equations.

We will adopt this approach and define in Figure 6 a fixpoint operator Φ for our semantical equations. The existence of a fixpoint for such an operator can be shown by exploiting a well-known result from functional analysis, namely the Brouwer-Schauder Theorem [27, Theorem 18.10].

Theorem 4.9 (Brouwer-Schauder Theorem) *Let $F : K \mapsto K$ be a continuous mapping from a non-empty closed, convex set K in a Banach space into itself, with the closure of $F(K)$ compact. Then there exists a fixed-point of F, i.e. a point $c \in K$ such that $F(c) = c$.*

This theorem although similar to the Banach's fixpoint theorem used in *metric semantics* [16] [6] differs from the latter for two important reasons: First, while Banach's theorem requires a *contractive* operator the requirement in the Brouwer-Schauder Theorem is weaker and consists essentially in the *non-expansiveness* of the operator F in as far as F operates on a closed, convex set K. Second, this is a non-constructive fixpoint theorem, i.e. it guarantees only the existence of a fixpoint, but does not say how to construct one nor guarantees its uniqueness, as Banach's theorem does.

It might be interesting to note that the proof of this theorem is based essentially on the "geometric" structure (convexity) of a vector space, which is not available in a simple metric space as in the Banach fixpoint theorem (cf. remarks in [27, pag. 188]).

In the following we will investigate the conditions in Theorem 4.9 in relation to our generic semantics in order to establish the existence of a semantics for the generic language in Figure 4.

4.2.1 The Effective Domain

Since C*algebras are special types of *Banach spaces* the choice of a C*algebra as the semantical domain for the operator Φ allows us to meet the first requirement on the domain in Theorem 4.9.

An important subset of a C*algebra is the *simplex* $\mathcal{S}$ which is defined as follows ([71, XII.1]):

Definition 4.10 Given a C*algebra $\mathcal{A}$ and a set of basic operators $\mathcal{B} = \{\mathbf{B}_i\}_i$. The *simplex* $\mathcal{S} = \mathcal{S}(\mathcal{B})$ generated by $\mathcal{B}$ is the closed convex hull of $\mathcal{B}$, i.e. the smallest closed subset of $\mathcal{A}$ containing all the convex combinations of elements in $\mathcal{B}$:

$$\sum_i \alpha_i \mathbf{B}_i$$

with $\alpha_i \geq 0$ and $\sum_i \alpha_i = 1$.

The simplex generated by the operators $\{\mathbf{B}_i\}_i$ corresponding to the basic actions and guards in the generic language is an appropriate candidate for the effective domain of the operator Φ, provided that Φ leaves $\mathcal{S}$ invariant, that is Φ maps elements in $\mathcal{S}$ into elements of $\mathcal{S}$. This condition is fulfilled whenever we ensure that the weighting in a probabilistic choice, $[]_{i=1}^n \; g_i \to p_i : A_i$ is normalised, i.e. $\sum_{i=1}^n p_i = 1$.

The fact that we model the parallel construct via the direct sum of all the possible interleavings of the sub-agents, makes it necessary the consideration of as many copies of the simplex $\mathcal{S}$ as the number of interleavings. Therefore, a more appropriate domain is the direct sum $\bigoplus_{i \in I} \mathcal{S}_i$, where I is an appropriate set of indices [6] and $\mathcal{S}_i = \mathcal{S}$ for all $i \in I$.

Definition 4.11 Consider a probabilistic instantiation $\mathcal{L}$ of the generic language. Let $\mathcal{A}$ be a C*algebra, and let $\mathcal{B} = \{\mathbf{B}_i\}_i \subseteq \mathcal{A}$ be the linear operators corresponding to the basic actions and guards of $\mathcal{L}$. We define the simplex $\mathcal{S}(\mathcal{L})$ generated by $\mathcal{L}$ as $\bigoplus_{i \in I} \mathcal{S}_i$, where I is a non-empty set of indices and $\mathcal{S}_i = \mathcal{S}(\mathcal{B})$ for all $i \in I$.

Proposition 4.12 *$\mathcal{S}(\mathcal{L})$ is a non-empty, closed, convex set in a Banach space.*

Proof. We assume that $\mathcal{B}$ is non-empty, therefore $\mathcal{S}(\mathcal{B})$ and thus $\mathcal{S}(\mathcal{L})$ are non-empty too. Furthermore, $\mathcal{S}(\mathcal{L})$ is the direct sum (i.e. essentially the disjoint union) of closed sets and therefore closed itself. Finally, $\mathcal{S}(\mathcal{L})$ is a convex set as it is the direct sum of convex sets. More precisely: Any element in $\mathcal{S}(\mathcal{L})$ is of the form: $x = \bigoplus_{i \in I} x_i$ with $x_i \in \mathcal{S}(\mathcal{B})$. Take the convex combination of any two elements x and y in $\mathcal{S}(\mathcal{L})$. We get, with $\lambda \in [0,1]$:

$$\lambda x + (1-\lambda) y = \lambda \left(\bigoplus_{i \in I} x_i \right) + (1-\lambda) \left(\bigoplus_{i \in I} y_i \right) = \bigoplus_{i \in I} \left(\lambda x_i + (1-\lambda) y_i \right).$$

For each $i \in I$ any convex combination of x_i and y_i — i.e. $\lambda x_i + (1-\lambda) y_i$ — is in $\mathcal{S}(\mathcal{B})$. We see therefore that for any $x, y \in \mathcal{S}(\mathcal{L})$ their convex combination $\lambda x + (1-\lambda) y$ is also in $\mathcal{S}(\mathcal{L})$. □

4.2.2 The Operator

In order to apply Theorem 4.9 we have to show that for a given probabilistic instantiation $\mathcal{L}$ of the generic language (1) $\mathcal{S}(\mathcal{L})$ is invariant under Φ, (2) Φ is continuous, and (3) the closure of $\Phi(\mathcal{S}(\mathcal{L}))$ is compact.

[6] I is essentially isomorphic to the set of all sequences of words or products of operators representing the basic actions and guards.

Invariance of $\mathcal{S}$:

The equations in Figure 5 and the fixpoint operator Φ in Figure 6 are formulated without specifying the basic operators $\mathbf{B}$ and guards $\mathbf{G}$. Depending on the concrete language, and its embedding into the generic language we have to fill in these "parameters". We will assume in the following that the basic operators are positive operators with $\|\mathbf{B}\| = 1$, and that the guards are represented by orthogonal projection operators: $\mathbf{G}^2 = \mathbf{G} = \mathbf{G}^*$. This guarantees that they form a suitable set of generators of a simplex.

Proposition 4.13 *Let $\mathcal{L}$ be a probabilistic instantiation of the generic language. Let $\mathcal{S}(\mathcal{L})$ be the simplex generated by $\mathcal{L}$. If $[\![\Box_{i=1}^{n}\ g_i \rightarrow p_i : A_i]\!]$ and $[\![\|_{i=1}^{n}\ p_i : A_i]\!]$ are normalised, that is $\sum_{i=1}^{n} p_i = 1$, then the operator Φ defined in Figure 6 maps elements of the simplex $\mathcal{S}(\mathcal{L})$ into elements in $\mathcal{S}(\mathcal{L})$, i.e.*

$$\Phi(\mathcal{S}(\mathcal{L})) \subseteq \mathcal{S}(\mathcal{L}).$$

Proof. By induction on the structure of the agent A and the fact that under the hypothesis that p_is are normalised the simplex $\mathcal{S}(\mathcal{L})$ is closed with respect to all the operations on the right hand side of the equations in Figure 6. □

Continuity of Φ:

The continuity of Φ follows from the following classical result [71,59]:

Theorem 4.14 *A linear map $\mathbf{T} : \mathcal{V} \mapsto \mathcal{W}$ from a normed vector space $(\mathcal{V}, \|.\|_{\mathcal{V}})$ into another normed vector space $(\mathcal{W}, \|.\|_{\mathcal{W}})$ is* continuous *if and only if it is bounded, i.e.*

$$\exists c \geq 0 : \|\mathbf{T}(x)\|_{\mathcal{W}} \leq c \cdot \|x\|_{\mathcal{V}}$$

Proposition 4.15 *The operator Φ defined in Figure 6 is continuous.*

Proof. It is clear that the operator Φ inductively defined in Figure 6 is *linear*. Furthermore by Proposition 4.13 it is bounded (as it leaves the simplex invariant). □

Compactness of Φ:

The condition that the closure of $\Phi(\mathcal{S})$ is compact can be equivalently formulated as the condition that $\Phi(\mathcal{S})$ is *relative compact*. Linear, bounded operators for which the image of the unit sphere (i.e. all operators with norm less or equal to one) is relative compact are called *compact* or completely continuous operators [71, X.2]. For our purposes it is therefore sufficient to show that Φ is a compact operator. In fact, since the simplex $\mathcal{S} = \mathcal{S}(\mathcal{B})$ is the closed convex hull of the operators $\mathcal{B}$, it can be shown that its image under a compact operator is also relative compact.

For the restricted case of a finite matrix algebra $\mathcal{L}(\mathbb{R}^n)$ it is easy to see that all operators in $\mathcal{L}(\mathbb{R}^n)$ are compact by using the Heine-Borel theorem [26]. This theorem ensures that every *closed* and *bounded* set in a finite dimensional vector space is *compact*, and the simplex $\mathcal{S}$ is one such set.

In the general case of infinite dimensional vector space, the compactness of the operator Φ follows by the following theorem (cf. [59, Thm VI.12], or [25].[71, X.2]):

Theorem 4.16

(a) *A linear combination of compact operators is compact.*

(b) *The product of a compact operator with a bounded linear operator is compact.*

Corollary 4.17 *Let $\mathcal{L}$ be a probabilistic instantiation of the generic language such that the number n in the choice and parallel constructs of $\mathcal{L}$ is finite (i.e. $\mathcal{L}$ is finitely branching). Consider the operator Φ defined in Figure 6. Assume that the basic operators $\mathcal{B}$ are compact, then Φ is compact.*

Proof. By Theorem 4.16. □

4.2.3 Existence of a Semantics

We can now show the existence of a solution to the semantical equations in Figure 5.

Theorem 4.18 *Consider a finitely branching probabilistic instantiation $\mathcal{L}$ of the generic language. Suppose that in the linear semantics the operators associated to the basic actions are compact operators in $\mathcal{S}(\mathcal{L})$ and the operators associated to the guards are projections. Then the operator Φ defined in Figure 6 has a fixpoint.*

Proof. By Proposition 4.12 the set $\mathcal{S}(\mathcal{L})$ is a non-empty, convex and closed subset of a Banach space. By Proposition 4.13 and Proposition 4.15 Φ is a continuous operator on $\mathcal{S}(\mathcal{L})$. Moreover, by Proposition 4.17 Φ is compact which implies that the image $\Phi(\mathcal{S}(\mathcal{L}))$ is relative compact. We can therefore apply the Brouwer-Schauder Theorem 4.9. □

5 Structural Complexity of the Linear Semantics

From a practical viewpoint, the linear semantics defined above is not necessarily an efficient encoding of the semantics of programming languages. On the contrary, the fact that the matrices and operators are highly sparse, i.e. contain mostly zero entries, suggests that in some cases (e.g. for analysis purposes) one might want to consider some more compact representation of the semantics. We investigate here the complexity of the structures effectively used in a concrete semantics. This depends substantially on the features of a specific language and in particular on the presence of synchronisation.

5.1 Independence and Commutativity

In analysing concurrent languages, one problem is related to the question of whether executing one agent A before another one B changes the behaviour of the second one. In general, if the order in which A and B are executed in a given context (i.e. a program with a placeholder $C[\cdot]$) does not change the final results, then the two agents are independent in the sense that one agent does not influence the behaviour of the other. We can formalise this notion by referring to the general language defined in Figure 4 and the notion of observables introduced in Section 3.3.2.

Definition 5.1

(a) Two agents A and B are *independent* iff $\mathcal{O}(C[(A \odot B)]) = \mathcal{O}(C[(B \odot A)])$, for all contexts C, i.e. the result of executing A and then B is the same as the result of executing B and then A in all contexts C.

(b) We call a language *synchronisation-free* if all its agents are independent.

Proposition 5.2 *Let A and B be two agents in the generic language such that $[\![A]\!] \cdot [\![B]\!] = [\![B]\!] \cdot [\![A]\!]$. Then A and B are independent.*

Proof. Consider a context C. Then

$$\begin{aligned}\mathcal{O}((C[A \odot B)]) &= K(N(I([\![C[(A \odot B)]]\!]))) \\ &= K(N(I([\![C[(B \odot A)]]\!]))) = \mathcal{O}(C[(B \odot A)]).\end{aligned}$$

□

In the case of a semantics which is fully abstract with respect to the observables we can show a stronger result.

Proposition 5.3 *Assume a linear semantics $[\![\cdot]\!]$ which is fully abstract with respect to the observables $\mathcal{O}$. Then the agents A and B are independent iff their semantics commute.*

Proof. Proposition 5.2 shows that $[\![A]\!] \cdot [\![B]\!] = [\![B]\!] \cdot [\![A]\!]$ implies the independence of A and B (for any correct semantics).

Suppose now that A and B independent agents. Then we have $\mathcal{O}(C[(A \odot B)]) = \mathcal{O}(C[(B \odot A)])$ for any context C. By the hypothesis that $[\![\cdot]\!]$ is fully abstract, we can conclude from the fact that the observables of $A \odot B$ and $B \odot A$ are the same in all contexts that $[\![A \odot B]\!] = [\![B \odot A]\!]$ and thus that the semantics of A and B commute:

$$[\![A]\!] \cdot [\![B]\!] = [\![A \odot B]\!] = [\![B \odot A]\!] = [\![B]\!] \cdot [\![A]\!].$$

□

For our notions of observables and linear semantics introduced in Section 3 we have shown that the linear operators for μ**PCLP** agents commute (Proposition 3.2. By using the representation theorems established in Section 4.1 we now show that the semantical domain for μ**PCLP** has a simple representation.

Proposition 5.4 *The linear semantics for μ**PCLP** agents can be expressed by means of continuous functions over a (locally) compact space.*

Proof. By Proposition 3.2 the linear semantics of the μ**PCLP** language forms a subset of an Abelian C*algebra. Then Theorem 4.8 applies. □

As shown in Examples 3.5 and 3.6, it is not possible to formulate a similar result for the linear semantics of μ**PCCP** and μ**PImp**. The presence of synchronisation forces the semantics of these languages to be expressed in terms of the more complex algebra of linear operators on a Hilbert space.

5.2 Declarativity: A Discussion

The role of commutativity in the formulation of a linear semantics for a language like μ**PCLP** reveals much of the essence of *declarative programming*.

Declarativity is usually intended as the property of a language which makes it possible for the programmer to concentrate on the logic of an algorithm. Thus, a program can be regarded as a logical theory and computations as deductions for proving that a given goal is a logical consequence of that theory. This makes the definition of an axiomatic semantics immediately available.

The fact that we can represent the linear semantics of a synchronisation-free language like μ**PCLP** in an algebra of continuous functions is an alternative, equivalent way to assert the declarativity of the language. In fact, an algebra of continuous functions essentially defines the topology of the underlying space $\mathcal{M}$ and vice versa (for a nice overview of the relation between algebraic and topological notions see e.g. [69, Sect 1.11]). By exploiting the Stone Duality, this topology can be directly translated into some "standard" (program) logic (cf. [1,2,3]).

For general, "non-commuting" languages this axiomatisation of the original linear semantics in terms of an essentially classical logic fails because we are unable to identify the semantics of agents with simple continuous functions. For these languages we are forced to stay within the more complex C*algebraic framework. There are various attempts to reconstruct "non-commutative" analogues to the topology of the Abelian case, among which we mention Mulvey's work on Quantales [52] and Connes's Non-Commutative Geometry [12,13].

5.3 The Finite Case

In a finite dimensional vector space, it is possible to illustrate the Abelian representation theorem by another well known theorem from linear algebra (cf. [58, Sect. 29]):

Theorem 5.5 *A set of commuting diagonalisable operators in a finite dimensional vector space V have a common eigenbasis.*

That means that we can diagonalise a set of diagonalisable operators *simultaneously*. A diagonalisable operator is an operator which can be transformed in an equivalent diagonal one. Roughly, this transformation exploits the eigenvalues of the operator with respect to some eigenbasis. In general, the diagonal form of an operator is obtained with respect to an eigenbasis which is different from the one which diagonalises an other operator. For commutative operators, however, it is possible to find a basis such that the representation of all of them is diagonal. This is in essence the finite-dimensional illustration of Gelfand's Duality Theorem.

The linear semantical operators we use for the semantics of probabilistic languages are in general not diagonalisable matrices. In fact, they are in general infinite operators rather than matrices. The case of finite matrices occurs when the State space is finite-dimensional and the language is synchronisation-free. In this case our matrices are still not diagonalisable in general. However, we believe that they can be simplified in a nearly diagonal form.

Every matrix can be transformed in an equivalent matrix which is nearly diagonal. This is called the Jordan Normal Form. Like for diagonalisation, it is in general not possible to find a common basis with respect to which a given set of operators can be 'Jordanised' simultaneously.

We conjecture that for the semantical operators of PCLP, this is possible. According to this conjecture, we would then be able to define the semantics of PCLP on $n < \infty$ states by means of nearly diagonal matrices. This would imply that we could reduce the number of free parameters from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. This conjecture is supported by the fact that in [21] we have defined a linear semantics for the full version of PCLP, whose linear operators are indeed vectors.

We have no conjectures for asserting that the complexity of the linear semantics for the imperative case, μ**PImp**, and the concurrent synchronised version of μ**PCCP** of μ**PCLP** can be established a priori to be less than $\mathcal{O}(n^2)$. Although it does not change the asymptotic complexity, we observe that the monotonicity of μ**PCCP** computations makes the actual dimension of the semantical matrices smaller than n^2. In fact, these matrices are upper-triangular, which reduces their dimension to $\frac{n(n+1)}{2}$.

6 Related Work

The work of Kozen on the semantics of a probabilistic imperative language [43,44] is closely related to our approach. In this work a Banach space is used as semantical domain for a denotational semantics. One main difference is that our language as well as our semantical treatment includes concurrency. Moreover, probabilities are introduced in our language at the syntactic level by means of the probabilistic choice construct, while Kozen's approach considers random assignments in the language syntax and random variables with their associated probability distribution, in the semantical model.

Another approach to the semantics of probabilistic languages has been pioneered in [62,36,37] and taken over in [39]. This approach is based on probabilistic powerdomain, an extension of the powerdomain construction with measure-theoretic concepts which allow for modelling probabilistic choice. This framework allows to accommodate uncountable domains, and seems therefore particularly suitable for functional programming.

An approach essentially different from ours as well as from the ones mentioned before is the weakest precondition semantics for probabilistic languages which has been extensively studied in [51,64,49]. It would be of great interest to relate our operator algebra based notions with this type of axiomatic semantics. This could be especially important in the context of the verification of probabilistic systems.

Finally, we would like to mention the work on probabilistic process algebras (e.g. [4,31,68,7], for an overview see [38]) which covers areas like probabilistic bisimulation [46,18], verification of probabilistic systems [5,33], and semantics of probabilistic systems [10,55,42].

7 Conclusions and Further Work

In this paper we have presented a generic approach to a denotational semantics for probabilistic concurrent languages which is based on linear structures. We have shown that linear operators, or more abstractly C*algebras, constitute an appropriate semantical domain for modelling both probabilistic and concurrent aspects of programming languages. The approach presented is generic in the sense that we consider a language scheme which can be instantiated to produce various specific probabilistic programming languages. As concrete instantiations of the generic language we have discussed three particular languages belonging to the *declarative* and the *imperative* programming paradigms. Nevertheless, other programming paradigms can be represented in our framework too.

The generic semantics given for the generic language can also be instantiated to produce an appropriate semantics for the specific languages. We have shown the existence of the generic semantics as the fixpoint of an appropriate semantical operator on the C*algebra representing the semantical domain. The effective construction of a linear semantics cannot be generalised as it depends on the specific features of the probabilistic language considered. For example, we have shown that for synchronisation-free languages the corresponding operators commute, which allows for a simpler representation of the semantics of the language in terms of continuous functions instead of operators. In the finite dimensional case, matrices can be replaced by vectors. For this case and in particular for the probabilistic constraint logic programming we discussed in this paper (although in its minimal form μ**PCLP**), the effective construction of the linear semantics has be achieved in [21] as the limit of a sequence of distributions representing special simplified form of linear operators.

On an operational level it is quite obvious how matrices can be utilised to encode transition graphs [8]. The extension of this "algebraic" approach to infinite graphs, resulting in Hilbert space operators, is also well established (see e.g. [50] or [11]). One main contribution of our approach is to use these results to re-formulate a transition system based semantics for probabilistic concurrent languages as a denotational (i.e. compositional and fixpoint) one.

The linear semantics approach relates the semantics of quantitative languages to various mathematical disciplines. The relation to stochastic and dynamic processes and systems is obvious: Our semantics can be seen as an encoding of the programs behaviour seen as a Markov Chain in terms of stochastic matrices. It is interesting to mention that a "de-probabilised" version of the theory of Markov Chains, sometimes known as Topological Markov Chains, plays an important role in Symbolic Dynamics [47], and that C*algebraic techniques are applied for classification purposes [45,14]. It would be interesting to re-formulate the essential parts of our quantitative framework in this qualitative setting, i.e. in terms of 0/1 matrices. For probabilistic languages we cannot avoid considering vector space like structures in order to accommodate the quantitative information represented by the probabilities. However, for the non-probabilistic languages, where we have as numerical domain only $\{0, 1\}$ it might be possible to reformulate the Gelfand–Naimark results (Theorem 4.7 and Theorem 4.8) in simpler, perhaps just combinatorial terms.

Finally, as discussed in Section 5.2, the representation of the semantics of non-concurrent languages via functions on a "nice" topological space allows — via the Stone Duality — for a study of the semantics of such languages using a classical or intuitionistic logic [1,35]. For the general *concurrent* case the only universal representation of a linear semantics we may achieve is in terms of bounded operators on a Hilbert space (Theorem 4.7). A logic able to analyse such operators must therefore reflect the geometric nature of structures like Hilbert spaces and C*algebras. One promising approach in this directions is via quantales [60].

References

[1] Abramsky, S., "Domain Theory and the Logic of Observable Properties," Ph.D. thesis, University of London (1987).

[2] Abramsky, S., "Domain Theory and the Logic of Observable Properties," Ph.D. thesis, University of London (1987).

[3] Abramsky, S., *Domain theory in logical form*, Annals of Pure and Applied Logic **51** (1991), pp. 1–77.

[4] Baeten, J., J. Bergstra and S. Smolka, *Axiomatizing probabilistic processes: Acp with generative probabilities*, in: W. Cleaveland, editor, *CONCUR '92*, Lecture Notes in Computer Science **630** (1992), pp. 472–485.

[5] Baier, C. and M. Kwiatkowska, *Model checking for a probabilistic branching time logic with fairness*, Technical Report CSR-96-12, School of Computer Science, University of Birmingham (1996).

[6] Bakker, J. and J. Rutten, editors, "Ten Years of Concurrency Semantics," World Scientific, Singapore, 1992.

[7] Bernardo, M. and R. Gorrieri, *A tutorial on empa: A theory of concurrent processes with nondeterminism, priorities, probabilities and time*, Technical Report UBLCS-96-17, Department of Computer Science, University of Bologna (1997).

[8] Biggs, N., "Algebraic Graph Theory," Cambridge Mathematical Library, Cambridge University Press, Cambridge, 1993, second edition.

[9] Bistarelli, S., U. Montanari and F. Rossi, *Semiring-based constraint satisfaction and optimization*, Journal of the ACM **44** (1997), pp. 201–236.

[10] Blute, R., J. Desharnais, A. Edalat and P. Panangaden, *Bisimulation for labelled Markov processes*, in: *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Warsaw, Poland, 1997, pp. 149–158.

[11] Chung, F. R., "Spectral Graph Theory," Regional Conference Series in Mathematics **92**, American Mathematical Society, Providence, Rhode Island, 1997.

[12] Connes, A., *Non-commutative differential geometry*, Publications Mathematiques IHES **62** (1986), pp. 41–144.

[13] Connes, A., "Noncommutative Geometry," Academic Press, San Diego, 1994.

[14] Cuntz, J. and W. Krieger, *A class of C^*-algebras and topological Markov chains*, Inventiones Mathematicae **56** (1980), pp. 251–268.

[15] Davidson, K. R., "C*-Algebras by Example," Fields Institute Monographs **6**, American Mathematical Society, Providence, Rhode Island, 1996.

[16] de Bakker, J. and E. de Vink, "Control Flow Semantics," Foundations of Computing Series, MIT Press, Cambridge, Massachusetts – London, England, 1996.

[17] de Boer, F. S., A. Di Pierro and C. Palamidessi, *Nondeterminism and Infinite Computations in Constraint Programming*, Theoretical Computer Science **151** (1995), pp. 37–78.

[18] de Vink, E. P. and J. J. Rutten, *Bisimulation for probabilistic transition systems: A coalgebraic approach*, Theoretical Computer Science **221** (1999), pp. 271–293.

[19] Di Pierro, A. and H. Wiklicky, *On probabilistic CCP*, in: A. Policriti, M. Falaschi and M. Navarro, editors, *Proceedings of APPIA-GULP-PRODE'97 Joint Conference on Declarative Programming*, 1997, pp. 225–234.

[20] Di Pierro, A. and H. Wiklicky, *An operational semantics for Probabilistic Concurrent Constraint Programming*, in: P. Iyer, Y. Choo and D. Schmidt, editors, *ICCL'98 – International Conference on Computer Languages* (1998), pp. 174–183.

[21] Di Pierro, A. and H. Wiklicky, *Probabilistic Concurrent Constraint Programming: Towards a fully abstract model*, in: L. Brim, J. Gruska and J. Zlatuska, editors, *MFCS'98 – Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **1450** (1998), pp. 446–455.

[22] Di Pierro, A. and H. Wiklicky, *Quantitative observables and averages in probabilistic constraint programming*, in: *New Trends in Constraints - Selected Papers of the ERCIM/Compulog Workshop on Constraints*, Lecture Notes in Artificial Intelligence **1865**, Berlin - Heidelberg - New York, 2000.

[23] Dixmier, J., "C^*-Algebras," North-Holland Mathematical Library **15**, North-Holland, Amsterdam – New York – Oxford, 1977.

[24] Doran, R. S., editor, "C^*-Algebras: 1943–1993 — A Fifty Year Celebration," Contemporary Mathematics **167**, American Mathematical Society, Providence, Rhode Island, 1994.

[25] Dunford, N. and J. T. Schwartz, "Linear Operators I: General Theory," Interscience – John Wiley & Sons, New York – Chicester, 1958.

[26] Engelking, R., "General Topology," Sigma Series in Pure Mathematics **6**, Heldermann Verlag, Berlin, 1989.

[27] Goebel, K. and W. Kirk, "Topics in Metric Fixed Point Theory," Cambridge studies in advanced mathematics **28**, Cambridge University Press, Cambridge, 1990.

[28] Greub, W. H., "Linear Algebra," Grundlehren der mathematischen Wissenschaften **97**, Springer Verlag, New York, 1967, third edition.

[29] Gupta, V., R. Jagadeesan and P. Panangaden, *Stochastic processes as concurrent constraint programs*, in: ACM, editor, *Symposium on Principles of Programming Languages (POPL)*, ACM SIGPLAN Notices (1999), pp. 189–202.

[30] Gupta, V., R. Jagadeesan and V. Saraswat, *Probabilistic concurrent constraint programming*, in: A. Mazurkiewicz and J. Winkowski, editors, *CONCUR '97: Concurrency Theory*, Lecture Notes in Computer Science **1243** (1997), pp. 243–257.

[31] Hillston, J., *PEPA: Performance enhanced process algebra*, Technical Report CSR-24-93, University of Edinburgh, Edinburgh, Scotland (1993).

[32] Hungerford, T. W., "Algebra," Graduate Texts in Mathematics **73**, Springer Verlag, New York – Berlin – Heidelberg, 1980.

[33] Huth, M. and M. Kwiatkowska, *On probabilistic model checking*, Technical Report CSR-96-15, School of Computer Science, University of Birmingham (1996).

[34] Jaffar, J. and J.-L. Lassez, *Constraint Logic Programming*, in: *Symposium on Principles of Programming Languages (POPL)*, 1987, pp. 111–119.

[35] Johnstone, P. T., "Stone Spaces," Cambridge University Press, Cambridge, 1982.

[36] Jones, C., "Probabilistic Non-Determinism," Ph.D. thesis, University of Edinburgh, Edingburgh (1993).

[37] Jones, C. and G. Plotkin, *A probabilistic powerdomain of evaluations*, in: *Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society Press, 1989, pp. 186–195.

[38] Jonsson, B., W. Yi and K. G. Larsen, "Probabilistic Extentions of Process Algebras," Elsivier Science, Amsterdam, 2001 pp. 685–710, see [**?**].

[39] Jung, A. and R. Tix, *The troublesome probabilistic powerdomain*, in: A. Edalat, A. Jung, K. Keimel and M. Kwiatkowska, editors, *Third Workshop on Computation and Approximation*, Electronic Notes in Theoretical Computer Science **13** (1998), p. 23.

[40] Kadison, R. V. and J. R. Ringrose, "Fundamentals of the Theory of Operator Algebras," Pure and Applied Mathematics **I** – **Elementary Theory**, Academic Press, New York – London, 1983.

[41] Kadison, R. V. and J. R. Ringrose, "Fundamentals of the Theory of Operator Algebras," Pure and Applied Mathematics **II** – **Advanced Theory**, Academic Press, Orlando – London, 1986.

[42] Katoen, J.-P., C. Baier and D. Latella, *Metric semantics for true concurrent real time*, Theoretical Computer Science **254** (2001), pp. 501–542.

[43] Kozen, D., *Semantics of probabilistic programs*, in: *20th Annual Symposium on Foundations of Computer Science* (1979), pp. 101–114.

[44] Kozen, D., *Semantics for probabilistic programs*, Journal of Computer and System Sciences **22** (1981), pp. 328–350.

[45] Krieger, W., *On dimension functions and topological Markov chains*, Inventiones Mathematicae **56** (1980), pp. 239–250.

[46] Larsen, K. G. and A. Skou, *Bisimulation through probabilistic testing*, Information and Computation **94** (1991), pp. 1–28.

[47] Lind, D. and B. Marcus, "An Introduction to Symbolic Dynamics and Coding," Cambridge University Press, Cambridge – New York – Melbourne, 1995.

[48] Marriott, K. and P. J. Stuckey, "Programming with Constraints: Am Introduction," The MIT Press, Cambridge, Massachusetts – London, England, 1998.

[49] McIver, A. and C. Morgan, *Probabilistic predicate transformers: Part 2*, Technical Report PRG-TR-5-96, Programming Research Group, Oxford University Computing Laboratory (1996).

[50] Mohar, B. and W. Woess, *A survey on spectra of infinite graphs*, Bulletin of the London Mathematical Society **21** (1988), pp. 209–234.

[51] Morgan, C., A. McIver, K. Seidel and J. Sanders, *Probabilistic predicate transformers*, Technical Report PRG-TR-4-95, Programming Research Group, Oxford University Computing Laboratory (1995).

[52] Mulvey, C. J., *&*, in: *Il Conveno di Topologia (Taormina)*, Rendiconti del Circolo Mathematico di Palermo, Palermo, 1986, pp. 94–104, serie II, nummero 12.

[53] Murphy, G. J., "C^*-Algebras and Operator Theory," Academic Press, San Diego, 1990.

[54] Nielson, H. R. and F. Nielson, "Semantics with Applications – A Formal Introduction," John Wiley & Sons, Chichester, 1992.

[55] Norman, G., *Metric semantics for reactive probabilistic processes*, Technical Report CSR-98-3, University of Birmingham, School of Computer Science (1998).

[56] Norris, J., "Markov Chains," Cambidge Series in Statistical and Probabilistic Mathematics, Cambridge University Press, Cambridge, 1997.

[57] Plotkin, G., *Domains*, Department of Computer Science, University of Edimburgh (1992), post-graduate lecture notes in advanced domain theory (incorporating the 'Pisa notes' 1981).

[58] Prasolov, V. V., "Problems and Theorems in Linear Algebra," Translation of Mathematical Monographs **134**, American Mathematical Society, Providence, Rhode Island, 1994.

[59] Reed, M. and B. Simon, "Methods of Modern Mathematical Physics I: Functional Analysis," Academic Press, Orlando, 1980, revised and enlarged edition.

[60] Rosenthal, K. I., *A note on Girard quantales*, Cahiers de Topologie et Geometrie Differentille Categoriques **XXXI** (1990), pp. 3–12.

[61] Rutten, J. and D. Turi, *Initial algebra and final coalgebra semantics for concurrency*, in: J. de Bakker et al., editors, *Proceedings of the REX workshop: A Decade of Concurrency – Reflections and Perspectives*, Lecture Notes in Computer Science **803** (1994), pp. 530–582.

[62] Saheb-Djahromi, N., *CPO's of measures for nondeterminism*, Theoretical Computer Science **12** (1980), pp. 19–37.

[63] Saraswat, V., M. Rinard and P. Panangaden, *Semantics foundations of concurrent constraint programming*, in: *Symposium on Principles of Programming Languages (POPL)* (1991), pp. 333–353.

[64] Seidel, K., C. Morgan and A. McIver, *An introduction to probabilistic predicate transformers*, Technical Report PRG-TR-6-96, Programming Research Group, Oxford University Computing Laboratory (1996).

[65] Seneta, E., "Non-negative Matrices and Markov Chains," Springer Verlag, New York – Heidelberg – Berlin, 1981, second edition.

[66] Thirring, W., "Lehrbuch der Mathematischen Physik 3: Quantenmechanik von Atomen und Molekülen," Springer Verlag, Wien – New York, 1979.

[67] Tijms, H. C., "Stochastic Models – An Algorithmic Approach," John Wiley & Sons, Chichester, 1994.

[68] van Glabbeek, R. J., S. A. Smolka and B. Steffen, *Reactive, generative and stratified models of probabilistic processes*, Information and Computation **121** (1995), pp. 59–80.

[69] Wegge-Olsen, N. E., "K-Theory and C*-Algebras — A Friendly Approach," Oxford University Press, Oxford – New York – Tokyo, 1993.

[70] Winskel, G., "The Formal Semantics of Programming Languages," MIT Press, Cambridge, Massachusetts – London, England, 1993.

[71] Yosida, K., "Functional Analysis," Springer Verlag, Berlin – Heidelberg – New York, 1980.

Appendix

A Basic Examples

We present some concrete examples which gives an idea of the type of mathematical objects involved in the linear semantics introduced in this paper.

A.1 μ**PCCP** *and* μ**PCLP**

The examples illustrating the semantics of μ**PCCP**/μ**PCLP** agents will be based on the simple constraint system $\mathcal{C}$ depicted in Figure A.1, which represents our set of states State.

$$\begin{tikzcd}
 & \mathit{false} = c \sqcup d \sqcup e \arrow[dl] \arrow[d] \arrow[dr] & \\
c \sqcup d \arrow[d] \arrow[dr] & c \sqcup e \arrow[dl] \arrow[dr] & d \sqcup e \arrow[dl] \arrow[d] \\
c \arrow[dr] & d \arrow[d] & e \arrow[dl] \\
 & \mathit{true} &
\end{tikzcd}$$

Fig. A.1. A Simple Constraint System.

We will assume the following enumeration of the constraints in $\mathcal{C}$:

$$\langle \mathit{true}, c, d, e, c \sqcup d, c \sqcup e, d \sqcup e, \mathit{false} \rangle .$$

These constraints represents the basis for the linear state space $\mathcal{V}(\mathcal{C})$, which is (isomorphic to) $\mathbb{R}^8$.

Thus, the semantics of μ**PCCP**/μ**PCLP** agents will be given in terms of matrices made up from 8×8 factor matrices.

A.1.1 Basic Operators
In $\mathcal{V}(\mathcal{C})$, the representation of the basic agents $\mathbf{tell}(c)$, and the guards $\mathbf{ask}(c)$ is given by the following matrices.

$$[\![\mathbf{tell}(c)]\!] = \begin{bmatrix} 0&1&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&0&1 \end{bmatrix} \quad \text{and} \quad [\![\mathbf{ask}(c)]\!] = \begin{bmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&1&1&0&0\\ 0&0&1&0&1&0&1&0\\ 0&0&0&1&0&1&1&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1 \end{bmatrix}$$

where, for example, $[\![\mathbf{tell}(c)]\!]_{3,5} = 1$ expresses the fact that starting from store d the execution of $\mathbf{tell}(c)$ will produce the store $c \sqcup d$, and $[\![\mathbf{ask}(c)]\!]_{2,2} = 1$ represents the fact that c is in the upward closure of c.

*A.1.2 μ**PCCP**/μ**PCLP** Programs*
Consider the following two agents A and B, which implements an unguarded choice:

$$\begin{aligned} A &\equiv \mathbf{tell}(c) \\ B &\equiv \mathbf{ask}(true) \;\rightarrow\; \tfrac{1}{2} : \mathbf{tell}(d) \\ &\quad\;\; [\!]\; \mathbf{ask}(true) \;\rightarrow\; \tfrac{1}{2} : \mathbf{tell}(e). \end{aligned}$$

We are interested in the parallel execution of these two agents,

$$\tfrac{1}{2} : A \;\|\; \tfrac{1}{2} : B.$$

The denotational semantics of this agent is the direct sum of the two interleavings $[\![A]\!][\![B]\!]$ and $[\![B]\!] \cdot [\![A]\!]$, where

$$\begin{aligned} [\![A]\!] &= [\![\mathbf{tell}(c)]\!] \\ [\![B]\!] &= \tfrac{1}{2}\, [\![\mathbf{tell}(d)]\!] \;+\; \tfrac{1}{2}\, [\![\mathbf{tell}(e)]\!]. \end{aligned}$$

Therefore the resulting matrix $[\![\frac{1}{2} : A \parallel \frac{1}{2} : B]\!]$ is:

$$\left[\begin{array}{cc}
\begin{matrix} 0&0&0&0&\frac{1}{4}&\frac{1}{4}&0&0\\ 0&0&0&0&\frac{1}{4}&\frac{1}{4}&0&0\\ 0&0&0&0&\frac{1}{4}&0&0&\frac{1}{4}\\ 0&0&0&0&0&\frac{1}{4}&0&\frac{1}{4}\\ 0&0&0&0&\frac{1}{4}&0&0&\frac{1}{4}\\ 0&0&0&0&0&\frac{1}{4}&0&\frac{1}{4}\\ 0&0&0&0&0&0&0&\frac{1}{2}\\ 0&0&0&0&0&0&0&\frac{1}{2} \end{matrix} & \\
 & \begin{matrix} 0&0&0&0&\frac{1}{4}&\frac{1}{4}&0&0\\ 0&0&0&0&\frac{1}{4}&\frac{1}{4}&0&0\\ 0&0&0&0&\frac{1}{4}&0&0&\frac{1}{4}\\ 0&0&0&0&0&\frac{1}{4}&0&\frac{1}{4}\\ 0&0&0&0&\frac{1}{4}&0&0&\frac{1}{4}\\ 0&0&0&0&0&\frac{1}{4}&0&\frac{1}{4}\\ 0&0&0&0&0&0&0&\frac{1}{2}\\ 0&0&0&0&0&0&0&\frac{1}{2} \end{matrix}
\end{array}\right]$$

Note that the factors in this matrix are identical. This stems from the fact that the matrices for $[\![A]\!]$ and $[\![B]\!]$ commute. This is because we are considering a μ**PCLP** (i.e. a synchronisation free) program.

The observables can be retrieved from the matrix obtained by averaging the two factors in $[\![\frac{1}{2} : A \parallel \frac{1}{2} : B]\!]$, namely:

$$\begin{bmatrix} 0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0\\ 0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0\\ 0&0&0&0&\frac{1}{2}&0&0&\frac{1}{2}\\ 0&0&0&0&0&\frac{1}{2}&0&\frac{1}{2}\\ 0&0&0&0&\frac{1}{2}&0&0&\frac{1}{2}\\ 0&0&0&0&0&\frac{1}{2}&0&\frac{1}{2}\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&0&1 \end{bmatrix}$$

which encodes, for example, the fact that if $\frac{1}{2} : A \parallel \frac{1}{2} : B$ is executed in the initial store *true* (corresponding to the first component) we expect as result a store containing with equal probability either $c \sqcup d$ (fifth coordinate) and $c \sqcup e$ (sixth coordinate).

Consider now a guarded version of the agents A and B defined above:

$$\begin{aligned} A &\equiv \mathbf{tell}(c) \\ B &\equiv \mathbf{ask}(\mathit{true}) \;\rightarrow\; \tfrac{1}{2} : \mathbf{tell}(d) \\ &\quad\; [\!]\; \mathbf{ask}(c) \;\rightarrow\; \tfrac{1}{2} : \mathbf{tell}(e), \end{aligned}$$

and their parallel composition $\frac{1}{2} : A \parallel \frac{1}{2} : B$.

The denotations of $\frac{1}{2} : A \parallel \frac{1}{2} : B$ are now

$$\begin{aligned} [\![A]\!] &= [\![\mathbf{tell}(c)]\!] \\ [\![B]\!] &= \tfrac{1}{2}\,[\![\mathbf{tell}(d)]\!] \;+\; \tfrac{1}{2}\,[\![\mathbf{ask}(c)]\!] \cdot [\![\mathbf{tell}(e)]\!]. \end{aligned}$$

Therefore, the matrix representing $[\![\frac{1}{2} : A \parallel \frac{1}{2} : B]\!]$ is the two factors direct sum:

$$\left[\begin{array}{cccccccc|cccccccc}
0&0&0&0&\frac14&\frac14&0&0 &&&&&&&& \\
0&0&0&0&\frac14&\frac14&0&0 &&&&&&&& \\
0&0&0&0&\frac14&0&0&\frac14 &&&&&&&& \\
0&0&0&0&0&\frac14&0&\frac14 &&&&&&&& \\
0&0&0&0&\frac14&0&0&\frac14 &&&&&&&& \\
0&0&0&0&0&\frac14&0&\frac14 &&&&&&&& \\
0&0&0&0&0&0&0&\frac12 &&&&&&&& \\
0&0&0&0&0&0&0&\frac12 &&&&&&&& \\
&&&&&&&& 0&0&0&0&\frac12&0&0&0 \\
&&&&&&&& 0&0&0&0&\frac14&\frac14&0&0 \\
&&&&&&&& 0&0&0&0&\frac12&0&0&0 \\
&&&&&&&& 0&0&0&0&0&0&0&\frac12 \\
&&&&&&&& 0&0&0&0&\frac14&0&0&\frac14 \\
&&&&&&&& 0&0&0&0&0&\frac14&0&\frac14 \\
&&&&&&&& 0&0&0&0&0&0&0&\frac12 \\
&&&&&&&& 0&0&0&0&0&0&0&\frac12
\end{array}\right]$$

Due to the non-commutativity introduced by the guard (which now projects out certain states), the two factors differ. The effect of B simply depends now on the fact that c has been told previously. Therefore, $[\![A]\!] \cdot [\![B]\!]$ and $[\![B]\!] \cdot [\![A]\!]$ are no longer the same matrix.

Again, the observables can be reconstructed by considering the following "compactified" matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & \frac{3}{4} & \frac{1}{4} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{3}{4} & 0 & 0 & \frac{1}{4} \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & \frac{3}{4} \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

which encodes, for example, the fact that if A and B are executed in parallel in the initial store *true* we expect $c \sqcup d$ as result three times as often as $c \sqcup e$ (with probability $\frac{3}{4}$ for the former as compared to $\frac{1}{4}$ for the latter).

*A.2 A μ**PImp** Example*

To keep the example as simple possible, we look at a μ**PImp** program which involves only two variables x_1 and x_2. Each variable can take one of the three values 0, 1, and 2.

The set of states State is therefore, as in standard approaches to denotational semantics, given by the set of function [Var $\mapsto$ Val] (where we consider here only total functions). The enumeration which we will use in the matrix representation is the following:

$$\begin{gathered} (x_1 = 0, x_2 = 0) \\ (x_1 = 0, x_2 = 1) \\ (x_1 = 0, x_2 = 2) \\ (x_1 = 1, x_2 = 0) \\ (x_1 = 1, x_2 = 1) \\ (x_1 = 1, x_2 = 2) \\ (x_1 = 2, x_2 = 0) \\ (x_1 = 2, x_2 = 1) \\ (x_1 = 2, x_2 = 2) \end{gathered}$$

Thus, the linear domain $\mathcal{V}(\text{State})$ is isomorphic to $\mathbb{R}^9$, and the semantics of an agent can be represented by 9×9 matrices (or more precisely by matrices build up from 9×9 factors, though we will not consider parallelism in the concrete examples below).

A.2.1 Basic Operators

The basic operation in μ**PImp** is the assignment. The following matrices encode the semantics of the assignments $x_1 := 2$, $x_2 := 1$ and $x_2 := 2$.

$$[\![x_1 := 2]\!] = \begin{bmatrix} 0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1 \end{bmatrix}$$

$$[\![x_2 := 1]\!] = \begin{bmatrix} 0&1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1&0 \end{bmatrix} \qquad [\![x_2 := 2]\!] = \begin{bmatrix} 0&0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&1 \end{bmatrix}$$

As an example of the encoding of a guard, consider the following matrix, which singles out those states where the first variable has an even value:

$$[\![even(x_1)]\!] = \begin{bmatrix} 1&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&1 \end{bmatrix}$$

A.2.2 Some Simple Programs

A very simple program in μ**PImp** is given by the statement

$$x_1 := 2;\ x_2 := 1.$$

The semantics of this program is obtained by multiplying the matrices corresponding to the two assignments $x_1 := 2$ and $x_2 := 1$:

$$\begin{bmatrix} 0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&1 \end{bmatrix} \cdot \begin{bmatrix} 0&1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0 \end{bmatrix} = \begin{bmatrix} 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0 \end{bmatrix}$$

Not surprisingly, we end up with certainty, i.e. probability 1 in the same state $(x_1 = 2, x_2 = 1)$, whatever state we started in.

Note that in this case, the matrices commute, i.e. we would get the same result by inverting the order of the two factors. This reflects the fact that the two assignments are independent. A different situation is when we combine assignments to the same variable. The semantics of $x_2 := 1;\ x_2 := 2$ is:

$$
\begin{bmatrix}
0&1&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0\\
0&0&0&0&1&0&0&0&0\\
0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0
\end{bmatrix}
\cdot
\begin{bmatrix}
0&0&1&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1
\end{bmatrix}
=
\begin{bmatrix}
0&0&1&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1
\end{bmatrix}
$$

i.e. we simply get the effect of $x_2 := 2$. Analogously, the semantics of $x_2 := 2;\ x_2 := 1$ is:

$$
\begin{bmatrix}
0&0&1&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1
\end{bmatrix}
\cdot
\begin{bmatrix}
0&1&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0\\
0&0&0&0&1&0&0&0&0\\
0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0
\end{bmatrix}
=
\begin{bmatrix}
0&1&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0\\
0&0&0&0&1&0&0&0&0\\
0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0
\end{bmatrix}
$$

i.e. identical to the matrix representing $x_2 := 1$.

A simple statement involving a choice is given by the following:

$$\frac{1}{2} : (x_2 := 1) \,\Box\, \frac{1}{2} : (x_2 := 2),$$

which is represented by the stochastic matrix:

$$\frac{1}{2}\begin{bmatrix} 0&1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 0&0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&0&0&1 \end{bmatrix} = \begin{bmatrix} 0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0\\ 0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0\\ 0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0\\ 0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0\\ 0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0\\ 0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0\\ 0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}\\ 0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}\\ 0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2} \end{bmatrix}$$

Finally, we can use the matrices defined before to give the semantics of slightly more "complex" statements, involving also the guard:

$$[\![(\mathbf{begin}(\mathbf{case}\frac{1}{2} : (x_2 := 1)\ \mathbf{or}; \frac{1}{2} : (x_2 := 2)))\ ;\ (\mathbf{if}\ even(x_1)\ \mathbf{then}\ (x_2 := 1))]\!] =$$

$$\begin{bmatrix} 0&1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0 \end{bmatrix}$$

Whenever this program terminates, the result is with certainty with a state with $x_2 = 1$. Such a final state is either $(x_1 = 0, x_2 = 1)$ (if we started in a state with $x_1 = 0$) or $(x_1 = 2, x_2 = 1)$ (if we started in $x_1 = 2$). If we start in a state with $x_1 = 1$ (i.e. x_1 not odd) the program will not terminate.

Analogously,

$$[\![(\mathbf{begin}(\mathbf{case}\frac{1}{2} : (x_2 := 1) \;\mathbf{or}\; \frac{1}{2} : (x_2 := 2))) \;;\; (\mathbf{if}\, even(x_1) \;\mathbf{then}\; (x_2 := 2))]\!] =$$

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2}
\end{bmatrix}$$

Whenever this agent terminates, the result is with probability 50% the state $(x_1 = 2, x_2 = 1)$ or with probability 50% the state $(x_1 = 2, x_2 = 2)$. Again, if we start in a state with $x_1 = 1$ (i.e. x_1 not odd) the agent will not terminate.

Observe that in both examples there is a set of states where no transitions happen (in the end) because the guard in the last statement is not satisfied. The computation gets stuck. One can think of several ways to overcome this situation. A discussion about these options is beyond the scope of this paper.