Probabilistic Modular Embedding for Stochastic Coordinated Systems

Stefano Mariani and Andrea Omicini

DISI, Alma Mater Studiorum–Università di Bologna
{s.mariani,andrea.omicini}@unibo.it

Abstract. *Embedding* and *modular embedding* are two well-known techniques for measuring and comparing the expressiveness of languages—sequential and concurrent programming languages, respectively. The emergence of new classes of computational systems featuring stochastic behaviours – such as pervasive, adaptive, self-organising systems – requires new tools for probabilistic languages. In this paper, we recall and refine the notion of *probabilistic modular embedding* (PME) as an extension to modular embedding meant to capture the expressiveness of stochastic systems, and show its application to different coordination languages providing probabilistic mechanisms for stochastic systems.

Keywords: embedding, modular embedding, coordination languages, probabilistic languages, π-calculus.

1 Introduction

A core issue for computer science since the early days, expressiveness of computational languages is essential nowadays, when the focus is shifting from sequential to concurrent languages. This holds in particular for coordination languages, which, by focussing on interaction, deal with the most relevant source of complexity in computational systems [1]. Unsurprisingly, the area of coordination models and languages has produced a long stream of ideas and results on the subject, both adopting/adapting "traditional" approaches – such as Turing equivalence for coordination languages [2,3] – and inventing its own original techniques [4].

Comparing languages based on either their structural properties or the observable behaviour of the systems built upon them is seemingly a good way to classify their expressiveness. Among the many available approaches, the notion of *modular embedding* [5], refinement of Shapiro's *embedding* [6], is particularly effective in capturing the expressiveness of concurrent and coordination languages. However, the emergence of classes of systems featuring new sorts of behaviours – pervasive, adaptive, self-organising systems [7,8] – is pushing computational languages beyond their previous limits, and asks for new models and techniques to observe, model and, measure their expressiveness. In particular, modular embedding fails in telling probabilistic languages apart from non-probabilistic ones.

Accordingly, in this paper we recall, refine, and extend applicability of the notion of *probabilistic modular embedding* (PME) sketched in [9] as a formal framework to capture the expressiveness of probabilistic languages and stochastic systems. After recalling the basis of embedding and modular embedding (ME), along with some formal tools for dealing with probability (Section 2), in Section 3 we devise out the requirements for a probabilistic framework, and formalise them so as to define the full notion of PME. In order to demonstrate its ability to measure language expressiveness, in Section 4 we test PME against ME using a number of different case studies – probabilistic coordination languages and calculi –, and show how PME succeeds where ME simply fails. After discussing related works, such as *bisimulation* and *probabilistic bisimulation* (Section 5), we provide some final remarks and hints at possible future works (Section 6).

2 Background

2.1 Sequential and Modular Embedding

The informal definition of *embedding* assumes that a language could be *easily* and *equivalently* translated in another one. "Easily" is usually interpreted as "without the need for a global reorganisation of the program", whatever this means; whereas "equivalently" typically means "without affecting the program's *observable behaviour*", according to some well-defined *observation criteria*, usually to be specified for the application at hand.

Such an intuitive definition was formalised by Shapiro [6] for *sequential languages* as follows. Given two languages L, L', their program sets $Prog_L, Prog_{L'}$, and the powersets of their observable behaviours Obs, Obs', we assume that two *observation criteria* Ψ, Ψ' hold:

$$\Psi : Prog_L \rightarrow Obs \qquad \Psi' : Prog_{L'} \rightarrow Obs'$$

Then, L *embeds* L' (written $L \succeq L'$) *iff* there exist a *compiler* $C : Prog_{L'} \rightarrow Prog_L$ and a *decoder* $D : Obs \rightarrow Obs'$ such that for every program $W \in L'$

$$D(\Psi[C(W)]) = \Psi'[W]$$

Subsequently, De Boer and Palamidessi [5] argued such definition to be too weak to be applied proficiently, because any pair of Turing-complete languages would embed each other. Moreover, *concurrent languages* need at least *(i)* a novel notion of *termination* w.r.t. sequential ones, so as to handle deadlock and computation failure, and *(ii)* a different definition for the compiler, so as to consider also a priori unknown *run-time interactions* between concurrent processes.

Following their intuitions, De Boer and Palamidessi proposed a novel definition of embedding for which C and D should satisfy three properties:

Independent Observation. Elements $O \in Obs$ are sets representing all the possible outcomes of all the possible computations of a given system, hence they will be typically observed independently one from the other—since they are different systems. Thus, D can be defined to be *elementwise*, that is:

$$\forall O \in Obs : D(O) = \{d(o) \mid o \in O\} \text{ (for some } d)$$

Compositionality of C. In a concurrent setting, it is difficult to predict the behaviour of all the processes in the environment, due to run-time non-deterministic interactions. Therefore, it is reasonable to require compositionality of the compiler C both w.r.t. the parallel composition operator ($||$) and to the exclusive choice ($+$). Formally:

$$C(A \;||'\; B) = C(A) \;||\; C(B) \quad \text{and} \quad C(A \;+'\; B) = C(A) \;+\; C(B)$$

for every pair of programs $A, B \in L'$, where $'$ denotes symbols of L'.

Deadlock Invariance. Unlike sequential languages, where only successful computations do matter – basically because unsuccessful ones could be supposed to backtrack –, in a concurrent setting we need to consider at least deadlocks, interpreting failure as a special case of deadlock, which should then be preserved by the decoder D:

$$\forall O \in \mathit{Obs}, \forall o \in O : \mathit{tm}'(D_{d(o)}) = \mathit{tm}(o)$$

where tm and tm' refer to termination modes of L and L' respectively.

If an embedding satisfies all the three properties above, then it is called *modular*. In the following, we stick with symbol $\succeq$ since we assume the corresponding notion as our "default" reference embedding.

2.2 Expressiveness of Modular Embedding

A Case Study: `ProbLinCa` In [10], the authors define the `ProbLinCa` calculus, a probabilistic extension of the `LinCa` calculus therein defined, too. Whereas the latter is basically a process-algebraic formalisation of standard Linda – accounting for `out`, `rd`, `in` primitives –, the former equips each tuple with a *weight*, resembling selection probability: the higher the weight of a tuple, the higher its probability to be selected for matching. Basically, when a tuple template is used in a Linda primitive, the weights of the (possibly many, and different in kind) matching tuples are summed up, then each (j) is assigned a probability value $p \in [0, 1]$ obtained as follows: $p_j = \frac{w_j}{\sum_{i=1}^{n} w_i}$.

Suppose the following `ProbLinCa` process P and `LinCa` process Q are acting on tuple space S:

$$P = \mathtt{in}_p(T).\emptyset + \mathtt{in}_p(T).\mathtt{rd}_p(T').\emptyset \qquad Q = \mathtt{in}(T).\emptyset + \mathtt{in}(T).\mathtt{rd}(T').\emptyset$$
$$S = \langle \mathtt{t_l}[20], \mathtt{t_r}[10] \rangle$$

where T is a Linda template matching both tuples $\mathtt{t_l}$ and $\mathtt{t_r}$, whereas T' matches $\mathtt{t_r}$ solely. Subscript p distinguishes a `ProbLinCa` primitive from a `LinCa` one; square brackets store the weight of each tuple. Let us suppose also that both processes have the following non-deterministic branching policy: branch left if consumption primitive ($\mathtt{in}_p$ and `in`) returns $\mathtt{t_l}$, branch right if consumption primitive returns $\mathtt{t_r}$—as subscript suggests.

From the *modular observable behaviour* viewpoint exploited in modular embedding (ME), P and Q are *not* distinguishable. In fact, according to any observation function Ψ defined based on [5], $\Psi[P] = \Psi[Q]$, that is, P and Q can reach the same final states:

$$\Psi[P] = (\texttt{success}, \langle \texttt{t}_\texttt{r}[10] \rangle) \text{ OR } (\texttt{deadlock}, \langle \texttt{t}_\texttt{l}[20] \rangle)$$
$$\Psi[Q] = (\texttt{success}, \langle \texttt{t}_\texttt{r}[10] \rangle) \text{ OR } (\texttt{deadlock}, \langle \texttt{t}_\texttt{l}[20] \rangle)$$

The main point here is that while P and Q are *qualitatively* equivalent, they are not *quantitatively* equivalent. Notwithstanding, by no means ME can distinguish between their behaviours: since ME cannot tell apart the probabilistic information conveyed by, e.g., a `ProbLinCa` primitive w.r.t. a `LinCa` one. For the don't know non-deterministic process Q, no *probability value* is available that could measure the chance to reach one state over the other, hence ME does not capture this property—quite obviously, since it was not meant to.

In fact, a "two-way" *modular encoding* can be trivially established between the two languages used above – say, `ProbLinCa` (`out`, $\texttt{rd}_p$, $\texttt{in}_p$) vs. `LinCa` (`out`, `rd`, `in`) – by defining compilers C as

$$C_{\texttt{LinCa}} = \begin{cases} \texttt{out} & \longmapsto \texttt{out} \\ \texttt{rd} & \longmapsto \texttt{rd}_p \\ \texttt{in} & \longmapsto \texttt{in}_p \end{cases} \quad C_{\texttt{ProbLinCa}} = \begin{cases} \texttt{out} & \longmapsto \texttt{out} \\ \texttt{rd}_p & \longmapsto \texttt{rd} \\ \texttt{in}_p & \longmapsto \texttt{in} \end{cases}$$

and letting decoder D be compliant to the concurrent notion of observables given in [5]. Given such C and D, we can state that `ProbLinCa` *("modularly") embeds* `LinCa` and also that `LinCa` (modularly) embeds `ProbLinCa`, hence they are *(observational) equivalent* ($\equiv_\Psi$). Formally:

$$\texttt{ProbLinCa} \succeq \texttt{LinCa} \ \wedge\ \texttt{LinCa} \succeq \texttt{ProbLinCa} \quad \Longrightarrow \quad \texttt{ProbLinCa} \equiv_\Psi \texttt{LinCa}$$

However, process P *becomes* a probabilistic process due to the weighted-probability feature of $\texttt{rd}_p$, hence a probabilistic measure for P behaviour would be potentially available: however, it is not captured by ME. In the above example, for instance, such an additional bit of information would make it possible to assess that one state is "twice as probable as" the other. This is exactly the purpose of the *probabilistic modular embedding* (PME) we refine and extend in the remainder of this paper.

2.3 Formal Tools for Probability: The "Closure" Operator

In order to better ground the notion of PME as an extension of ME, and also to show its application to probabilistic coordination languages such as the aforementioned `ProbLinCa`, a proper formal framework is required.

In [11] a novel formalism is proposed that aims at dealing with the issue of open transition systems specification, requiring *quantitative information* to be attached to synchronisation actions at run-time—that is, based on the *environment* state during the computation. The idea is that of *partially closing* labelled transition systems via a process-algebraic *closure* operator ($\uparrow$), which associates quantitative values – e.g., probabilities – to admissible transition actions based upon a set of *handles* defined in an application-specific manner, dictating which quantity should be attached to which action. More precisely:

(i) actions labelling open transitions are equipped with handles;
(ii) the operator $\uparrow$ is exploited to compose a system to a specification G, associating at run-time each handle to a given value—e.g., a value $\in \mathbb{N}$;
(iii) quantitative informations with which to equip actions – e.g., probabilities $\in [0, 1]$ summing up to 1 – are computed from handle values for each enabled action, possibly based on the action context (environment);
(iv) quantitatively-labelled actions turn an open transition into a reduction, which then executes according to the quantitative information.

For instance, such operators are used as *restriction* operators in the case of `ProbLinCa` (Subsection 3.2) – in the same way as [12] for PCCS (Probabilistic Calculus of Communicating Systems) – to formally define the probabilistic interpretation of *observable actions*, informally given in Subsection 3.1, in the context of a tuple-based probabilistic language. In particular:

(i) handles coupled to actions (open transition labels) represent tuple templates associated to corresponding primitives;
(ii) handles listed in restriction term G represent tuples offered (as synchronisation items) by the tuple space;
(iii) restriction term G associates handles (tuples) to their weight in the tuple space;
(iv) restriction operator $\uparrow$ matches admissible synchronisations between processes and the tuple space, cutting out unavailable actions, and computes their associated probability distribution based upon handle-associated values.

3 Probabilistic Modular Embedding

3.1 Probabilistic Setting Requirements

In order to define the notion of *probabilistic modular embedding* (PME), we elaborate on the notion sketched in [9], starting from the informal definition of "embedding", then giving a precise characterisation to both words "easily" and "equivalently". Although the definition of "easily" given in Subsection 2.1 could be rather satisfactory in general, we prefer here to strengthen its meaning by narrowing its scope to asynchronous coordination languages and calculi: without limiting the generality of the approach, this allows us to make precise assumptions on the structure of programs.

A process can be said to be *easily* mappable into another if it requires:

(i) no extra-computations to mimic complex coordination operators;
(ii) no extra-coordinators (neither coordinated processes nor coordination medium) to handle suspensive semantics;
(iii) no unbounded extra-interactions to perform additional coordination.

Requirement *(i)* ensures absence of internal protocols in-between process-medium interactions, to emulate complex interaction primitives or behaviours—e.g., $\texttt{in}_p$

probabilistic selection simulated by processes drawing random numbers. Requirement *(ii)* avoids proliferation of processes and media while translating a program into another, constraining such mappings to have the same number of processes and media. The last requirement complements the first in ensuring absence of complex interaction patterns to mimic complex coordination operators, such as the `in_all` global primitive as a composition of multiple `inp` (`in` predicative version)—which could be obtained by forbidding unbounded *replication* and *recursion* algebraic operators in compiler C. Altogether, the three requirements above represent a necessary constraint since our goal here is to focus on "pure coordination" expressiveness—that is, we intentionally focus on the sole expressiveness of coordination primitives, while abstracting away from the *algorithmic expressiveness* of processes and media.

The refined notion of "equivalently" is a bit more involved due to the very nature of a probabilistic process, that is, its intrinsic *randomness*. The notions of *observable behaviour* and *termination* are affected by such randomness, thus they could need to be re-casted in the probabilistic setting. Probabilistic processes, in fact, have their actions conditioned by probabilities, hence their observable transitions between reachable states are probabilistic, too—so, their execution is possible but never guaranteed. Therefore, also final states are reached by chance only, following a certain *probability path*, hence termination, too, should be equipped with its own associated probability—also in case of deadlock.

In order to address the above issues, PME improves ME by making the following properties about observable behaviour and termination available:

Probabilistic Observation. Observable actions performed by processes – e.g., `ProbLinCa` coordination primitives – should be associated with their *execution probability*. Such probability should depend on their run-time context, that is, synchronisation opportunities offered by the coordination medium. Then, compiler C should preserve transition probabilities and properly "aggregate" them along any *probabilistic trace*—that is, a sequence of probabilistic actions.

Probabilistic Termination. Final states of processes and media should be first defined as those states for which all outgoing transitions have probability 0. Furthermore, they should be refined with a *probabilistic reachability value*, that is, the probability of reaching that state from a given initial one. Then, decoder D should preserve such probabilities and determine how to compute them.

3.2 Formal Semantics

In the following, we provide a precise semantic characterisation for the PME requirements pointed out in Subsection 3.1 – that is, *probabilistic observation* and *probabilistic termination* – using `ProbLinCa` primitives as our running example, which may also be regarded as a generalisation of uniform primitives used in [9].

Probabilistic Observation. A single probabilistic observable transition step, deriving from the synchronisation between a `ProbLinCa` process and a `ProbLinCa` space – e.g. by using a $\mathtt{in}_p$ –, can be formally defined as follows:

$$\mathtt{in}_p(T).P \mid \langle t_1[w_1], .., t_n[w_n]\rangle \quad \xrightarrow{\mu(T,t_j)}_{p_j} \quad P[t_j/T] \mid \langle t_1[w_1], .., t_n[w_n]\rangle \backslash t_j$$

where operator $\mu(T,t)$ denotes LINDA matching function, symbol $[\cdot/\cdot]$ stands for template substitution in process continuation, and operator $\backslash$ represents multiset difference, there expressing removal of tuple t_j from the tuple space.

By expanding such observable transition in its embedded reduction steps – that is, non-observable, silent transitions – we can precisely characterise the probabilistic semantics thanks to the $\uparrow$ operator:

$$\begin{gathered}
\mathtt{in}_p(T).P \mid \langle t_1[w_1], .., t_n[w_n]\rangle \\
\xrightarrow{T} \\
\mathtt{in}_p(T).P \mid \langle t_1[w_1], .., t_n[w_n]\rangle \uparrow \{(t_1,w_1), .., (t_n,w_n)\} \\
\hookrightarrow \\
\mathtt{in}_p(T).P \mid \langle t_1[w_1], .., t_n[w_n]\rangle \uparrow \{(t_1,p_1), .., (t_j,p_j), .., (t_n,p_n)\} \\
\xrightarrow{t_j}_{p_j} \\
P[t_j/T] \mid \langle t_1[w_1], .., t_n[w_n]\rangle \backslash t_j
\end{gathered}$$

where $p_j = \frac{w_j}{\sum_{i=1}^{n} w_i}$ is the absolute probability of retrieving tuple t_j (with $j = 1..n$) assuming for the sake of simplicity that all tuples match template T.

The $\uparrow$ operator implicitly enforces a re-normalisation of probabilities based on available synchronisations offered by the tuple space – that is, which tuples in the space match the given template –, in the spirit of PCCS restriction operator used in [12] for the generative model of probabilistic processes. For instance, given the following *probabilistic* process P acting on `ProbLinCa` space S

$$P = \tfrac{1}{6}\mathtt{in}_p(T).P + \tfrac{1}{2}\mathtt{in}_p(T).P + \tfrac{1}{3}\mathtt{rd}_p(T).P$$
$$S = \langle \mathtt{t_l}[w], \mathtt{t_r}[w]\rangle$$

where template T matches with both tuples $\mathtt{t_l}$, $\mathtt{t_r}$, we observe that an "experiment" $\mathtt{in}_p$ could succeed with probability $\frac{2}{3} = \frac{1}{6} + \frac{1}{2}$, and that $\mathtt{rd}_p$ could do so with probability $\frac{1}{3}$. Furthermore, if we suppose the branching choice after experiment $\mathtt{in}_p$ to depend upon the consumed tuple, we could see the aforementioned re-normalisation by computing the probability to branch left ($\mathtt{t_l}$ is returned), which is 0.25, and that of branching right ($\mathtt{t_r}$ is returned), which is instead 0.75.

This addresses the first issue of probabilistic observation: we define *observable actions* as all those actions requiring synchronisation with the medium, then equip them with a probability of execution driven by available run-time synchronisation opportunities, and normalised according to the generative model interpretation enforced by $\uparrow$. Formally, we define the *probabilistic observation function* (Θ), mapping a process (W) into observables, as follows:

$$\Theta[W] = \Big\{(\rho, W[\bar{\mu}]) \mid \quad (W, \langle\sigma\rangle) \longrightarrow^{*} (\rho, W[\bar{\mu}])\Big\}$$

where ρ is a probability value $\in [0, 1]$, $\bar{\mu}$ is a sequence of actual synchronisations – e.g. $\bar{\mu} = \mu(T_1, t_1), \ldots, \mu(T_n, t_n)$ – and σ is the space state—e.g. $\sigma = t_1, \ldots, t_n$.

Such definition is partial in the sense that we only know how to compute ρ for single-step transitions – that is, according to the $\uparrow$-dependent generative semantics –, in fact, it tells us nothing about how to compute ρ also for *observable traces*—that is, for sequences of observable actions. Nothing more than standard probability theory is needed here, stating that [13]:

(i) the cumulative probability of a *sequence* – that is, a *"dot"-separated* list – of probabilistic actions is the *product* of the probabilities of such actions;
(ii) the cumulative probability of a *choice* – that is, a *"+"-separated* list – of probabilistic actions is the *sum* of the probabilities of such actions.

Formally, we define the *sequence probability aggregation function* ($\bar{\nu}$) and the *choice probability aggregation function* (ν^+), mapping multiple probability values to a single one, as follows:

$$\bar{\nu} : W \times \langle\sigma\rangle \mapsto \rho \;\text{ where }\; \rho = \prod_{j=0}^{n}\{p_j \mid (p_j, \mu_{\bar{\ell}}) \in \Theta[W = \bar{\ell}.W'])\}$$
$$\nu^+ : W \times \langle\sigma\rangle \mapsto \rho \;\text{ where }\; \rho = \sum_{j=0}^{n}\{p_j \mid (p_j, \mu_{\ell^+}) \in \Theta[W = \ell^+.W'])\}$$

where $\bar{\ell}$ is a *sequence* of synchronisation actions – e.g., $\bar{\ell} = \mathtt{in}_p(T_1).\mathtt{rd}_p(T_2).\ldots$ – and ℓ^+ is a *choice* between synchronisation actions—e.g., $\bar{\ell}^+ = \mathtt{in}_p(T_1) + \mathtt{rd}_p(T_2) + \ldots$. By properly composing such aggregation functions, it is possible to compute $\Theta[W]$ for any process W and for any transition sequence $\longrightarrow^*$.

An example may help clarifying the above definitions. Let us consider the following process P and space S (sequence operator has priority w.r.t. choice):

$$P = \mathtt{in}_p(T).\big(\mathtt{rd}_p(T').P' + \mathtt{rd}_p(T').P''\big) + \mathtt{in}_p(T).P'$$
$$S = \langle \mathtt{t_{l1}}[\mathtt{40}], \mathtt{t_{l2}}[\mathtt{30}], \mathtt{t_{r1}}[\mathtt{20}], \mathtt{t_{r2}}[\mathtt{10}]\rangle$$

where template T may match either $\mathtt{t_{l1}}$ or $\mathtt{t_{r1}}$ whereas T' may match either $\mathtt{t_{l2}}$ or $\mathtt{t_{r2}}$—and branching structure is based on returned tuple as usual, that is, $\mathtt{t_{l1}}$,$\mathtt{t_{l2}}$ for left, $\mathtt{t_{r1}}$,$\mathtt{t_{r2}}$ for right. Applying function Θ to process P could lead to the following *observable states*:

$$\Theta[P] = (0.5, P'[\mu(T, \mathtt{t_{l1}}), \mu(T', \mathtt{t_{l2}})])$$
$$\Theta[P] = (0.1\bar{6}, P''[\mu(T, \mathtt{t_{l1}}), \mu(T', \mathtt{t_{r2}})])$$
$$\Theta[P] = (0.\bar{3}, P'[\mu(T, \mathtt{t_{r1}})])$$

According to Θ, and using both aggregation functions $\bar{\nu}$ and ν^+, we can state that process P will eventually behave like P' – although with different substitutions – with a probability of $\simeq 0.83$, and like P'' with a probability of $\simeq 0.17$.

As a last note, one may consider that sequence probability aggregation function $\bar{\nu}$ will asymptotically tend to 0 as the length of the sequence $\bar{l}$ tends to infinity. This is unavoidable according to the basic probability theory framework adopted throughout this paper. One way to fix this aspect could be that of considering only the prefix sequence executed in loop by a process, then to associate that process not with the probability of n iterations of such loop, but with the probability of the looping prefix sequence solely—that is, with only 1 iteration of the loop. However, this concern is left for future investigation.

Probabilistic Termination. In order to define *probabilistic termination*, we should first adapt the classical notion of termination to the probabilistic setting. For this purpose, we define *ending states* as all those states for which either no more transitions are possible or all outgoing transitions have probability 0 to occur. Other than that, termination states can be enumerated as usual [5] to be $\tau = \texttt{success}, \texttt{failure}, \texttt{deadlock}$, plus the `undefined` state, which could be useful to distinguish *absorbing states* – that is, those states for which the probability of performing a self-loop transition is 1 – from deadlocks. Furthermore, such termination states have to be equipped with a *probabilistic reachability value*, according to Subsection 3.1.

Formally, we define reachability value $\rho_\perp$ and the *probabilistic termination state function* Φ as follows:

$$\Phi[W] = \Big\{ (\rho_\perp, \tau) \mid \quad (W, \langle\sigma\rangle) \longrightarrow^*_\perp (\rho_\perp, \tau) \Big\}$$

where subscript $\perp$ means a sequence of finite transitions leading to termination τ. By comparing this function with the observation function Θ, it can be noticed that Φ abstracts away from computation *traces* – that is, it does not keep track of synchronisations, hence substitutions, in term $W[\bar{\mu}]$ – focussing solely on termination states τ. However, when computing the value of $\rho_\perp$, the same aggregation functions $\bar{\nu}$ and ν^+ have to be used.

For instance, recalling process P used to test observation function Θ, changing space S configuration as follows:

$$P = \texttt{in}_p(T).\big(\texttt{rd}_p(T').P' + \texttt{rd}_p(T').P''\big) + \texttt{in}_p(T).P'$$
$$S = \langle \texttt{t}_{\texttt{l1}}[40], \texttt{t}_{\texttt{r1}}[20] \rangle$$

where $P' \equiv P'' = \emptyset$ so to reach termination, the application of the probabilistic termination state function just defined would lead to the following *observable termination states*:

$$\Phi[P] = (0.\bar{6}, \texttt{deadlock}) \quad \vee \quad \Phi[P] = (0.\bar{3}, \texttt{success})$$

In particular, P deadlocks with probability $\frac{2}{3}$ if tuple $\texttt{t}_{\texttt{l1}}$ is consumed, whereas succeeds with probability $\frac{1}{3}$ if tuple $\texttt{t}_{\texttt{r1}}$ is consumed in its stead.

Please notice that, unlike the case of endless sequences $\bar{l}$ highlighted at the end of Subsection 3.2, absorbing states cause no harm for our sequence probability aggregation function $\bar{\nu}$, since the probability value aggregated until reaching the absorbing state will be from now on always multiplied by 1—in the very end, making each iteration of the self-loop indistinguishable from the others.

4 PME vs. ME: Testing Expressiveness on Case Studies

`ProbLinCa` vs. `LinCa` We now recall the two processes P and Q acting on space S introduced in the example of Subsection 2.2:

$$P = \texttt{in}_p(T).\emptyset + \texttt{in}_p(T).\texttt{rd}_p(T').\emptyset \qquad Q = \texttt{in}(T).\emptyset + \texttt{in}(T).\texttt{rd}(T').\emptyset$$
$$S = \langle \texttt{t}_{\texttt{l}}[20], \texttt{t}_{\texttt{r}}[10] \rangle$$

to repeat the embedding observation, this time under the assumptions of PME. As expected, we can now distinguish the *behaviour* of process P from that of process Q. In fact, by applying function Φ to both P and Q we get:

$$\Phi[P] = (0.\bar{6}, \texttt{success}) \text{ OR } (0.\bar{3}, \texttt{deadlock})$$
$$\Phi[Q] = (\bullet, \texttt{success}) \text{ OR } (\bullet, \texttt{deadlock})$$

where symbol $\bullet$ denotes "absence of information".

Therefore, only a "one-way" *encoding* can be now established between the two languages used above – again, `ProbLinCa` (`out`, $\texttt{rd}_p$, $\texttt{in}_p$) vs. `LinCa` (`out`, `rd`, `in`) – by defining compiler $C_{\texttt{LinCa}}$ as

$$C_{\texttt{LinCa}} = \begin{cases} \texttt{out} & \longmapsto & \texttt{out} \\ \texttt{rd} & \longmapsto & \texttt{rd}_p \\ \texttt{in} & \longmapsto & \texttt{in}_p \end{cases}$$

and making decoder D rely on observation function Θ and termination function Φ. Then we can state that `ProbLinCa` *probabilistically embeds* ($\succeq_p$) `LinCa`—but not the other way around. Formally, according to PME:

$$\texttt{ProbLinCa} \succeq_p \texttt{LinCa} \;\wedge\; \texttt{LinCa} \not\succeq_p \texttt{ProbLinCa} \quad \Longrightarrow \quad \texttt{ProbLinCa} \not\equiv_p \texttt{LinCa}$$

In the end, PME succeeds in telling `ProbLinCa` apart from `LinCa` (classifying `ProbLinCa` as *more expressive* than `LinCa`), whereas ME fails.

pKlaim vs. Klaim pKlaim was introduced in [14] as a probabilistic extension to Klaim [15], a kernel programming language for mobile computing. In Klaim, processes as well as data can be moved across the network among computing environments: in fact, it features *(i)* a core Linda with multiple tuple spaces, and *(ii) localities* as first-class abstractions to explicitly manage mobility and distribution-related aspects. pKlaim extends such model by introducing probabilities in a number of different ways and according to the two levels of Klaim formal semantics: *local* and *network* semantics. We here consider local semantics solely, because the network one can quickly become cumbersome, due to multiple probability normalisation steps [15], and is unnecessary for the purpose of showing the power of the PME approach.

The local semantics defines how a number of co-located processes interact with a tuple space, either local or remote. Here, probabilities are given by:

- a probabilistic choice operator $+_{i=1}^{n} p_i : P_i$;
- a probabilistic parallel operator $|_{i=1}^{n}\, p_i : P_i$;
- *probabilistic allocation environments*, formally defined as a partial map $\sigma : \mathit{Loc} \mapsto \mathit{Dist}(S)$ associating probability distributions on physical sites (S) to logical localities (Loc).

For the sake of clarity (and brevity), we consider the three probabilistic extensions introduced by pKlaim separately—their combination is a trivial extension once that normalisation procedures [15] are accounted for.

First of all, we focus on the probabilistic choice operator. Let us suppose pKLAIM process P and KLAIM process Q are interacting with space s—refer to [14] for process syntax:

$$\begin{gathered} P = \tfrac{2}{3}\texttt{in}(T)@s.\emptyset + \tfrac{1}{3}\texttt{in}(T)@s.\texttt{rd}(T)@s.\emptyset \\ Q = \texttt{in}(T)@s.\emptyset + \texttt{in}(T)@s.\texttt{rd}(T)@s.\emptyset \\ s = \texttt{out}(\texttt{t})@\texttt{self}.\emptyset \quad \equiv \quad s = \langle \texttt{t} \rangle \end{gathered}$$

where T matches the single tuple $\texttt{t}$ and KLAIM notation for tuple space s is equivalent to our usual notation.

Both processes have a non-deterministic branching structure which cannot be distinguished by ME. In fact, according to any observation function Ψ defined based on [5], $\Psi[P] = \Psi[Q]$, that is, P and Q can reach the same final states:

$$\begin{gathered} \Psi[P] = (\texttt{success}, \langle\,\rangle) \text{ OR } (\texttt{deadlock}, \langle\,\rangle) \\ \Psi[Q] = (\texttt{success}, \langle\,\rangle) \text{ OR } (\texttt{deadlock}, \langle\,\rangle) \end{gathered}$$

PME is instead sensitive to the probabilistic information available for pKLAIM process P, hence by applying the Φ function we get:

$$\begin{gathered} \Phi[P] = (0.\bar{6}, \texttt{success}) \text{ OR } (0.\bar{3}, \texttt{deadlock}) \\ \Phi[Q] = (\bullet, \texttt{success}) \text{ OR } (\bullet, \texttt{deadlock}) \end{gathered}$$

Since the probabilistic parallel operator can be handled (almost) identically, we step onwards to the probabilistic allocation operator. Suppose, then, to be in the following network configuration:

$$\begin{gathered} P = \texttt{in}(T)@l.\emptyset \qquad Q = \texttt{in}(T)@l.\emptyset \\ s_1 = \langle \texttt{t} \rangle \quad s_2 = \langle\,\rangle \qquad \sigma : l \mapsto \begin{cases} \frac{2}{3}s_1 \\ \frac{1}{3}s_2 \end{cases} \end{gathered}$$

where function σ is defined only for the pKLAIM process P—whereas for KLAIM process Q the allocation function is unknown (e.g., implementation-dependent). Thus, the processes have is branching structure and are actually identical. However, the coexistence of two admissible allocation environments (s_1 and s_2) may impact the termination states of P and Q. In fact, by applying any observation function suitable to ME criteria [5], we get the following final states:

$$\begin{gathered} \Psi[P] = (\texttt{success}, s_1 = \langle\,\rangle \wedge s_2 = \langle\,\rangle) \text{ OR } (\texttt{deadlock}, s_1 = \langle \texttt{t} \rangle \wedge s_2 = \langle\,\rangle) \\ \Psi[Q] = (\texttt{success}, s_1 = \langle\,\rangle \wedge s_2 = \langle\,\rangle) \text{ OR } (\texttt{deadlock}, s_1 = \langle \texttt{t} \rangle \wedge s_2 = \langle\,\rangle) \end{gathered}$$

Nevertheless, final states are the same for both P and Q, which cannot be distinguished. This happens because ME is *insensitive* to the probabilistic allocation function σ, having no ways to account for it.

PME provides instead "probability-sensitive" observation/termination functions, whose application produces the following final states:

$$\begin{gathered} \Phi[P] = (0.\bar{6}, \texttt{success}) \text{ OR } (0.\bar{3}, \texttt{deadlock}) \\ \Phi[Q] = (\bullet, \texttt{success}) \text{ OR } (\bullet, \texttt{deadlock}) \end{gathered}$$

Unlike ME, PME tells pKlaim process P apart from Klaim process Q. Since the difference between P and Q is quantitative, not qualitative, only a quantitative embedding such as PME can successfully distinguish between the two.

***π_{pa}-calculus* vs. *π_a-calculus*.** The π_{pa}-*calculus* was introduced in [16] as a probabilistic extension to the π_a-*calculus* (asynchronous π-*calculus*) defined in [17]. In order to increase the expressive power of π_a-*calculus*, the authors propose a novel calculus, featuring a probabilistic guarded choice operator ($\sum_i p_i \alpha_i.P_i$), able to distinguish between probabilistic and purely non-deterministic behaviours. Whereas the former is due to a random choice performed by the process itself, the latter is associated to the arbitrary decisions made by an external process (scheduler).

Let us consider the following processes P and Q willing to synchronise with process S (playing the role of the tuple space)—refer to [16] for the syntax used:

$$P = \big(\tfrac{2}{3}x(y) + \tfrac{1}{3}z(y)\big).\emptyset \qquad Q = \big(x(y) + z(y)\big).\emptyset$$
$$S = \bar{x}y \quad \equiv \quad S = \{S_x = \langle y \rangle \bigcup S_z = \langle\ \rangle\}$$

where the last equivalence just aims at providing a uniform notation compared to the other case studies proposed—in particular, π-*calculus* channels can resemble Klaim allocation environments.

For both P and Q, two are the admissible termination states, listed below as a result of the application of ME:

$$\Psi[P] = (\texttt{success}, \langle\ \rangle) \text{ OR } (\texttt{deadlock}, \langle y \rangle)$$
$$\Psi[Q] = (\texttt{success}, \langle\ \rangle) \text{ OR } (\texttt{deadlock}, \langle y \rangle)$$

As expected, they are indistinguishable despite the probabilistic information available for P, lost by the ME observation function.

As in previous cases, PME fills the gap:

$$\Phi[P] = (0.\bar{6}, \texttt{success}) \text{ OR } (0.\bar{3}, \texttt{deadlock})$$
$$\Phi[Q] = (\bullet, \texttt{success}) \text{ OR } (\bullet, \texttt{deadlock})$$

5 Related Works

One of the starting points of our work is the observation that modular embedding (ME) as defined in [5] does not suit probabilistic scenarios, since it does not consider probabilistic transitions—therefore probabilistic termination. In other words, when applied to a probabilistic process, ME can just point out its reachable termination states and the admissible transitions, and cannot say anything about their *quantitative* aspects—that is, probability of execution (transitions) and reachability (end states). Thus, in this paper we discussed how probabilistic modular embedding (PME) extends ME towards probabilistic models and languages, such as those defined in [12].

To the best of our knowledge, no other researches push the work by De Boer and Palamidessi [5] towards a probabilistic setting with a focus on coordination languages, in the same way as no other works try to connect the concept of "language embedding" with any of the probability models defined in [12].

In [18], the authors try to answer questions such as how to formalise probabilistic transition system, and how to extend non-probabilistic process algebras operators to the probabilistic setting. In particular, they focus on reactive models of probability – hence, models where pure non-determinism and probability coexist – and provide the notions of *probabilistic bisimulation*, *probabilistic simulation* (the asymmetric version of bisimulation), and *probabilistic testing preorders* (testing-based observation of equivalence), again applied to PCCS. Although targeted to the PCCS reactive model, their work is related to ours in the attempt to find a way to *compare* the relative expressiveness of different probabilistic languages. On the other hand, our approach is quite different because we adopted a *language embedding* perspective rather than a *process bisimulation* viewpoint. Whereas probabilistic bisimulation can prove the *observational equivalence* of different probabilistic models, it cannot detect which is the most expressive among them. However, we cannot exclude that a "two-way" probabilistic embedding relationship may correspond to a probabilistic bisimulation according to [18] definition of bisimulation—at least for reactive models.

In [19] the notion of *linear embedding* is introduced. Starting from the definition of ME in [5], the authors aim at *quantifying* "how much a language embeds another one", that is, "how much a given language is more expressive than another". To do so, they *(i)* take *linear vector spaces* as a semantic domain for a subset of LINDA-like languages – that is, considering `tell, get, ask, nask` primitives –; *(ii)* define an observation criteria associating to each program a linear algebra operator acting on such vector spaces; then *(iii)* quantify the difference in expressive power by computing the *dimension* of the linear algebras associated to each language. Although the possibility to *quantify* the relative expressive power of a set of languages is appealing, the work in [19] do consider neither probabilistic languages nor probabilistic processes, hence cannot be directly compared to ours. However, it still remains an interesting path to follow for further developments of the probabilistic embedding here proposed.

Last but not least, in [20] the authors apply the *Probabilistic Abstract Interpretation* (PAI) theory and its techniques to probabilistic transition systems, in order to formally define the notion of *approximate process equivalence*—that is, two probabilistic processes are equivalent "up to an error ε". As in [19], Di Pierro, Hankin, and Wiklicky adopt linear algebras to represent some semantical domain, but they consider probabilistic transition systems instead of deterministic ones. Therefore, they allow matrices representing algebraic operators to specify probability values $v \in [0, 1]$ instead of binary values $b = 0 \mid 1$. Then, by using the PAI framework and drawing inspiration from *statistical testing* approaches, they define the notion of *ε-bisimilarity*, which allows the minimum number of tests needed to accept the bisimilarity relation between two processes to be quantified with a given confidence. By examining such a number, a quantitative idea of the *statistical distance* between two given sets of (processes) admissible behaviours can be inferred. Although quite different from ours, this work can be considered nevertheless as another opportunity for further improvement of PME: for instance, an enhanced version of PME may be able to detect some notion of approximate process equivalence.

6 Conclusion and Future Works

Starting from the notions of embedding and modular embedding, in this paper we refine and extend the definition of *probabilistic modular embedding* (PME) first sketched in [9], as a tool for modelling the expressiveness of concurrent languages and systems, in particular those featuring probabilistic mechanisms and exhibiting stochastic behaviours. We discuss its novelty with respect to the existing approaches in the literature, then show how PME succeeds in telling apart probabilistic languages from non-probabilistic ones, whereas standard ME fails. While apparently trivial, such a distinction was not possible with any other formal framework in the literature so far, to the best of our knowledge.

Furthermore, PME has seemingly the potential to compare the expressiveness of two probabilistic languages: for instance, the ability of PME of telling apart the different probabilistic processes models proposed in [12] is currently under investigation.

Acknowledgments. This work has been partially supported by the EU-FP7-FET Proactive project SAPERE – Self-aware Pervasive Service Ecosystems, under contract no. 256874.

References

1. Wegner, P.: Why interaction is more powerful than algorithms. Communications of the ACM 40(5), 80–91 (1997)
2. Denti, E., Natali, A., Omicini, A.: On the expressive power of a language for programming coordination media. In: 1998 ACM Symposium on Applied Computing (SAC 1998), February 27-March 1, pp. 169–177. ACM, Atlanta (1998); Special Track on Coordination Models, Languages and Applications
3. Busi, N., Gorrieri, R., Zavattaro, G.: On the expressiveness of Linda coordination primitives. Information and Computation 156(1-2), 90–121 (2000)
4. Wegner, P., Goldin, D.: Computation beyond Turing machines. Communications of the ACM 46(4), 100–102 (2003)
5. de Boer, F.S., Palamidessi, C.: Embedding as a tool for language comparison. Information and Computation 108(1), 128–157 (1994)
6. Shapiro, E.: Separating concurrent languages with categories of language embeddings. In: 23rd Annual ACM Symposium on Theory of Computing (1991)
7. Omicini, A., Viroli, M.: Coordination models and languages: From parallel computing to self-organisation. The Knowledge Engineering Review 26(1), 53–59 (2011); Special Issue 01 (25th Anniversary Issue)

8. Omicini, A.: Nature-inspired coordination models: Current status, future trends. ISRN Software Engineering 2013, Article ID 384903, Review Article (2013)
9. Mariani, S., Omicini, A.: Probabilistic embedding: Experiments with tuple-based probabilistic languages. In: 28th ACM Symposium on Applied Computing (SAC 2013), Coimbra, Portugal, March 18-22, pp. 1380–1382 (2013) (Poster Paper)
10. Bravetti, M., Gorrieri, R., Lucchi, R., Zavattaro, G.: Quantitative information in the tuple space coordination model. Theoretical Computer Science 346(1), 28–57 (2005)
11. Bravetti, M.: Expressing priorities and external probabilities in process algebra via mixed open/closed systems. Electronic Notes in Theoretical Computer Science 194(2), 31–57 (2008)
12. van Glabbeek, R.J., Smolka, S.A., Steffen, B.: Reactive, generative, and stratified models of probabilistic processes. Information and Computation 121(1), 59–80 (1995)
13. Drake, A.W.: Fundamentals of Applied Probability Theory. McGraw-Hill College (1967)
14. Di Pierro, A., Hankin, C., Wiklicky, H.: Probabilistic KLAIM. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 119–134. Springer, Heidelberg (2004)
15. De Nicola, R., Ferrari, G., Pugliese, R.: KLAIM: A kernel language for agent interaction and mobility. IEEE Transaction on Software Engineering 24(5), 315–330 (1998)
16. Herescu, O.M., Palamidessi, C.: Probabilistic asynchronous pi-calculus. CoRR cs.PL/0109002 (2001)
17. Boudol, G.: Asynchrony and the Pi-calculus. Rapport de recherche RR-1702, INRIA (1992)
18. Bengt, J., Larsen, K.G., Yi, W.: Probabilistic extensions of process algebras. In Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra. Elsevier Science B.V., pp. 685–710 (2001)
19. Brogi, A., Di Pierro, A., Wiklicky, H.: Linear embedding for a quantitative comparison of language expressiveness. Electronic Notes in Theoretical Computer Science 59(3), 207–237 (2002); Quantitative Aspects of Programming Languages (QAPL 2001 @ PLI 2001)
20. Di Pierro, A., Hankin, C., Wiklicky, H.: Quantitative relations and approximate process equivalences. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 508–522. Springer, Heidelberg (2003)