From Distributions to Probabilistic Reactive Programs

Riccardo Bresciani and Andrew Butterfield[⋆]

Foundations and Methods Group,
Trinity College Dublin,
Dublin, Ireland
{bresciar,butrfeld}@scss.tcd.ie

Abstract. We have introduced probability in the *UTP* framework by using functions from the state space to real numbers, which we term *distributions*, that are embedded in the predicates describing the different program constructs. This has allowed us to derive a probabilistic theory of designs starting from a probabilistic version of the relational theory, and continuing further down this road we can get to a theory of probabilistic reactive programs. This paper presents the route that connects these steps, and discusses the challenges lying ahead in view of a probabilistic *CSP* based on distributions.

1 Introduction

The Unifying Theories of Programming (*UTP*) aims at a semantic framework where programs and specifications can be modelled as alphabetised relational predicates, capturing the semantic models normally used for their formal description [HH98, DS06, But10, Qin10]: the advantage of this common framework is that of enabling formal reasoning on the integration of the different languages through untyped predicate calculus.

So far several theories have been given a *UTP* semantics, where programs are expressed by means of logical predicates (programs are predicates! [Heh84, Hoa85]).

In the last years the focus of our research has been how to integrate probability into the *UTP* framework: our approach is based on distributions over the state space. We use distributions to associate a probability with each state: a program can therefore be expressed by means of logical predicates involving a homogeneous relation between distributions, to account for the modifications transforming *before-distributions* into corresponding *after-distributions*. This approach gives us a framework where probabilistic choice co-exists with non-deterministic choice, so being consistent with the approach advocated in [MM04].

After having given a probabilistic *UTP* semantics to *pGCL* [BB11, BB12b] and having presented a probabilistic theory of designs [BB12a], we have started

⋆ This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero — the Irish Software Engineering Research Centre.

to look into the possibility of using our framework to have a probabilistic version of *CSP*: as the *UTP* theory of *CSP* is built on that of designs, we aim at building a theory of *pCSP* starting from that of probabilistic designs. The task turned out to be not so straightforward, posing interesting challenges which we find worthy of discussion in the present paper.

This paper is structured as follows: we describe the background to UTP, with particular focus on the standard theory of designs in that framework, and to *pCSP* (§2); introduce our probabilistic framework based on distributions over the state space (§3.1), with a brief presentation of the probabilistic theory of designs from [BB12a] (§3.2); we then discuss how to progress from this probabilistic theory of designs to a theory of reactive programs (§4); and conclude (§5).

2 Background

2.1 UTP

UTP uses second-order predicates to represent programs: they are used to express relations among a set of *observable variables* which constitute their alphabet. Observable variables usually occur as both undecorated and decorated with a dash $'$: the former refer to states before the program starts (*before-states*), whereas the latter refer to the final states reached after the program has run (*after-states*). For example, a program using two variables x and y might be characterised by having the set $\{x, x', y, y'\}$ as an alphabet, and the meaning of the assignment $x := y + 3$ would be described, in a simple relational theory, by the predicate

$$x' = y + 3 \wedge y' = y.$$

In effect *UTP* uses predicate calculus in a disciplined way to build up a relational calculus for reasoning about programs.

In addition to observations of the values of program variables, often we need to introduce observations of other aspects of program execution via so-called auxiliary variables. For example the theory of reactive programs explained below uses four auxiliary variables — namely $ok, wait, tr, ref$ — to keep track of information concerning the current program run, such as termination, reach of a stable state, refusals, ...

A key notion in *UTP* is that of *healthiness conditions*: they are usually characterised as monotonic idempotent predicate transformers whose fixpoints characterise sensible (healthy) predicates. In other words they outlaw some predicates that are nonsense, e.g., $\neg ok \Rightarrow ok'$, which describes a "program" that must terminate even though not started.

This notion is closely related to that of *refinement*, defined as the universal closure[1] of reverse implication:

$$S \sqsubseteq P \mathrel{\hat=} [P \Rightarrow S]$$

[1] Square brackets denote universal closure, *i.e.* $[P]$ asserts that P is true for all values of its free variables.

Healthy predicates form a lattice under the ordering induced by the refinement relation. The refinement calculus enables the derivation of an implementation P from a specification S: such derivation can be proven correct if P is a valid refinement of S.

Some lines of research, including ours, are moving in the direction of introducing a probabilistic choice operator, which does not replace Dijkstra's demonic choice [Dij76] — as for example Kozen did [Koz81, Koz85] —, but rather coexists with it, as described and motivated in [MM04]. In [HS06] the authors present an approach to unification of probabilistic choice with standard constructs, and present an axiomatic semantics to capture *pGCL* in *UTP*: the laws were justified via a Galois connection to an expectation-based semantic model. The approach presented in [CS09] is that of decomposing non-deterministic choice into a combination of pure probabilistic choice and a unary operator that accounted for its non-deterministic behaviour. It is worth underlining a comment of theirs, on how UTP theories are still unsatisfactory with respect to the issue of having co-existing probabilistic and demonic choice. The *UTP* model described in [He10], which is used to give a *UTP*-style semantics to a probabilistic BPEL-like language, relates an initial state to a final probability distribution over states.

Our approach is a *UTP*-style semantics based on predicates over probability before- and after-distributions: we see programs as *distribution-transformers* (more details in §3.1). We have previously used this to encode the semantics of *pGCL* in the *UTP* framework [BB11, BB12b]; moreover we have proposed a probabilistic theory of designs [BB12a], which we will briefly present in §3.2 after having presented the standard one.

The Standard Theory of Designs. Now that we have given a general overview of the *UTP* framework, we are going to focus on the theory of designs and present its *UTP* semantics.

The theory of designs extends the simple (relational) theory, which is only adequate for partial correctness results, into a theory of total correctness. The motivation for and details of this extension are discussed in [HH98, Chapter 3]. This extension adopts an additional "auxiliary variable" ok (along with its dashed version ok') to record start (and termination) of a program. So now, instead of just observing variable values, we can now tell when a programs has been started, or has finished.

A design (specification) consists of a precondition $\mathcal{P}re$ that has to be met when the program starts, and if so the program terminates and establishes $\mathcal{P}ost$, which can be stated as:

$$ok \wedge \mathcal{P}re \Rightarrow ok' \wedge \mathcal{P}ost$$

for which we use the following shorthand:

$$\mathcal{P}re \vdash \mathcal{P}ost$$

Note that, in general, the "pre-condition" $\mathcal{P}re$ can mention after-values of variables and the "post-condition" $\mathcal{P}ost$ can mention before-values. The usual usage

H1	:	$P = (ok \Rightarrow P)$	(unpredictability)
H2	:	$P\{false/ok'\} \Rightarrow P\{true/ok'\}$	(possible termination)
H3	:	$P ; Skip = P$	(dischargeable assumptions)
H4	:	$\exists ok', \underline{v}' \bullet P$	(feasibility)

Fig. 1. Design Healthiness Conditions

of designs is however to restrict the pre-conditions to only refer to the before-values of variables. The semantics of the assignment $x := y + 3$ in this theory is the following:

$$true \vdash x' = y + 3 \wedge y' = y$$

(if started, it will terminate, and the final value of x will equal the initial value of y plus three, with y unchanged).

Designs form a lattice w.r.t. the refinement ordering, whose bottom and top elements are respectively *Abort* and *Miracle*:

$$\begin{array}{rcccl} Abort & \widehat{=} & false \vdash P & \equiv & true, \quad \text{for any predicate } P \\ Miracle & \widehat{=} & true \vdash false & \equiv & \neg ok \end{array}$$

It should be noted that *Miracle* is a (infeasible) program that cannot be started.

There are four healthiness condition associated with designs, called H1 through H4 (see Fig. 1). The first two characterise predicates that are designs (i.e., predicates that can be written in the form $P \vdash Q$), whilst the third restricts designs to those whose pre-condition does not mention after-observations (it is defined using *Skip* which is described later). The first three, either individually or combined, define sublattices with *Abort* and *Miracle* as extremal values. The fourth healthiness condition rules out infeasible predicates, such as *Miracle*, but breaks the lattice structure (it removes the top element, at least). Some of these conditions can be characterised by algebraic laws (H3 is defined that way):

$$\begin{array}{lll} \text{H1} \;\; true ; P = true & \text{and} & Skip ; P = P \\ \text{H4} \;\; P ; true = true & & \end{array}$$

In §3.2 we present a probabilistic version of this theory based on our framework.

CSP in the UTP Framework. Reactive programs differ from ordinary sequential programs because observing them in just the initial and final states is no longer sufficient, as there are some observable intermediate steps that characterise their behaviour, i.e., their interactions with the environment. In addition to observations ok and ok', which now correspond to a process being divergence-free, we add three more observations:

$wait, wait'$	:	$\mathbb{B}$	— waiting to perform an event
tr, tr'	:	$Event$-seq	— history of events being performed (trace)
ref, ref'	:	$Event$-set	— events currently not allowed (refusals)

$$\begin{array}{llll}
\mathsf{R1} & : & P = P \land (tr \le tr') & \text{(no time travel)} \\
\mathsf{R2} & : & P = \exists s \bullet P[s, s \frown (tr' - tr)/tr, tr'] & \text{(no direct event memory)} \\
\mathsf{R3} & : & P = \mathit{II} \lhd wait \rhd P & \text{(say nothing until started)} \\
 & : & \mathit{II} \mathrel{\widehat=} (\neg ok \land tr \le tr') \lor (ok' \land tr' = tr \land wait' = wait \land ref' = ref) & \\
\mathsf{R} & : & \bigwedge_{i \in 1,2,3} \mathsf{Ri} &
\end{array}$$

Fig. 2. Reactive Healthiness Conditions

$$\begin{aligned}
Stop &= \mathsf{R}(true \vdash ok' \land wait' \land tr' = tr) \\
Skip &= \mathsf{R}(true \vdash ok' \land \neg wait' \land tr' = tr) \\
a \to Skip &= \mathsf{R}(true \vdash (ok' \land (a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle)))
\end{aligned}$$

Fig. 3. Reactive Design semantics of CSP primitives

There are a number of associated healthiness conditions (Fig. 2). The first (R1) outlaws time travel by insisting that after-traces tr' are extensions of before-traces tr, whilst the second (R2) outlaws a process from having a direct memory of past events (any history-dependent behaviour requires some explicit state to remember some abstraction of past events). The third condition (R3) captures the fact that when not started, because some prior process is waiting ($wait = true$), we simply reflect the current behaviour of the prior process. This is captured by predicate II which requires us to propagate observations faithfully if the previous process is stable ($ok = true$). If the prior process has diverged ($ok = false$), then all we can guarantee is R1.

Originally, the theory of *communicating sequential processes* (*CSP*) was defined by adding in CSP-specific healthiness conditions CSP1–CSP2 [HH98, Chapter 8]. However a key unification result allows us to characterise CSP-healthy processes as *Reactive Designs* [OCW09]:

$$\mathsf{R}(\mathcal{P}re \vdash \mathcal{P}ost)$$

In other words, any CSP process can be written in the form of a design, that is "made" reactively healthy.

The semantics of some CSP constructs in this style are shown in Fig. 3.

2.2 pCSP

In §4 we are going to discuss how to create a *UTP*-friendly probabilistic variant of *CSP*. Here we look at two pieces of work regarding probabilistic *CSP*, that discuss some issues which are addressed by our theory.

In [MMSS96] we can find one of the possible definitions of *pCSP*, where probability is defined in such a way that it distributes through all operators. This leads to the surprising result that the demonic choice operator is not idempotent.

A refinement operator is defined, and the ideas of an associated probabilistic refinement calculus are discussed, where an implementation satisfies a specification with null probability. In effect we can no longer show whether an implementation satisfies a specification, but rather have to give bounds on the probability that an implementation may fail. This probability should ideally be very low, but would be expected to rise over time. In effect we have an implementation whose correctness has a life-time with some expectation value.

A different presentation is given in [Mor04], where *pCSP* is built on top of probabilistic action systems written in *pGCL* and is linked back to the relational semantics of *pGCL*. This view of the subject highlights how compositionality of probabilistic *CSP* is not straight-forward, because of the introduction of probability. Introducing probability splits a deterministic case into several possible different scenarios, and one has to take this into account when composing probabilistic programs.

They explain this using the metaphor of the colour of a child's eye: knowing the colour of the parents' eyes is not sufficient to predict that of the child. Instead we need hidden information about the alleles present and their relative dominance. In a similar fashion, in order to get accurate probabilities associated with *pCSP*, we have to track hidden information about choices that occurred in the past history.

3 Probabilistic Designs

3.1 The Distributional Framework

We are going to introduce briefly the key elements and constructs that characterise our distributional framework, in order to provide the reader with a working knowledge of it: a formal and rigorous definition can be found in [BB11], along with some soundness proofs.

Our framework relies on the concept of *distributions* over the state space, real-valued functions $\chi : \mathcal{S} \to \mathbb{R}$ that assign a *weight* x_i (a real number) to each state σ_i in the state space $\mathcal{S}$. We note the set of distributions as $\mathcal{D}$.

A state $\sigma : \mathcal{V} \to \mathcal{W}$ is a finite map from variables ($\mathcal{V}$) to values ($\mathcal{W}$). Each distribution has a weight, defined as:

$$\|\chi\| \mathrel{\widehat{=}} \sum_{\sigma \in \operatorname{dom} \chi} \chi(\sigma)$$

Among all distributions we distinguish *weighting distributions* π, such that $0 \le \pi(\sigma) \le 1$ for any state, and *probability (sub-)distributions* δ, such that $\|\delta\| \le 1$.

Generally speaking, it is possible to operate on distributions by lifting pointwise operators such as addition, multiplication and multiplication by a scalar[2]. Analogously we can lift pointwise all traditional relations and functions on real numbers.

[2] Distributions form a vector space, which we have explored elsewhere[BB11]. We omit discussion of this aspect of our theory for clarity and brevity.

In the case of pointwise multiplication, it is interesting to see it as a way of "re-weighting" a distribution. We have a particular interest in the case when one of the operands is a weighting distribution π, as we will use this operation to give semantics to choice constructs. We opt for a postfix notation to write this operation, as this is an effective way of marking when pointwise multiplication happens in the operational flow: for example if we multiply the probability distribution δ by the weighting distribution π, we write this as $\delta\langle\pi\rangle$. We use notation ϵ and ι to denote the everywhere zero and unit distributions, respectively:

$$\epsilon(\sigma) = 0 \wedge \iota(\sigma) = 1, \qquad \text{for all } \sigma$$

Given a condition (predicate on state) c, we can define the weighting distribution that maps every state where c evaluates to true to 1, and every other state to 0: we overload the above notation and note this distribution as $\iota\langle c\rangle$. In general whenever we have the multiplication of a distribution by $\iota\langle c\rangle$, we can use the postfix operator $\langle c\rangle$ for short, instead of using $\langle\iota\langle c\rangle\rangle$. It is worth pointing out that if we multiply a probability distribution δ by $\iota\langle c\rangle$, we obtain a distribution whose weight $\|\delta\langle c\rangle\|$ is exactly the probability of being in a state satisfying c.

Assignment. Given a simultaneous assignment $\underline{v} := \underline{e}$, where underlining indicates that we have lists of variables and expressions of the same length, we denote its effect on an initial probability distribution δ by $\delta\{\!|\underline{e}/\underline{v}|\!\}$. The postfix operator $\{\!|\underline{e}/\underline{v}|\!\}$ reflects the modifications introduced by the assignment — the intuition behind this, roughly speaking, is that all states σ where the expression $\underline{e}$ evaluates to the same value $\underline{w} = \mathrm{eval}_\sigma(\underline{e})$ are replaced by a single state $\sigma' = (\underline{v} \mapsto \underline{w})$ that maps to a probability that is the sum of the probabilities of the states it replaces.

$$(\delta\{\!|\underline{e}/\underline{v}|\!\})(\sigma') \mathrel{\hat=} (\textstyle\sum \delta(\sigma) \;\mid\; \sigma' = \sigma \dagger \{\underline{v} \mapsto \mathrm{eval}_\sigma(\underline{e}))$$

Here we treat the state as a map, where $\dagger$ denotes map override; this operator essentially implements the concept of "push-forward" used in measure theory, and is therefore a linear operator. An example is given in Figure 4.

Assignment preserves the overall weight of a probability distribution if $\underline{e}$ can be evaluated in every state, and if not the assignment returns a sub-distribution, where the "missing" weight accounts for the assignment failing on some states (this failure prevents a program from proceeding and causes non-termination).

Programming Constructs. The semantic definitions of various programming constructs are based on a homogeneous relation between distributions and are listed in Figure 5; we will now proceed to discuss each one.

The failing program Abort is represented by the predicate $\|\delta'\| \leq \|\delta\|$, which captures the fact that it is maximally unpredictable. However it is still guaranteed that the distribution weight cannot be increased, because that describes a program whose probability of termination is higher than that of it starting, and this is clearly impossible.

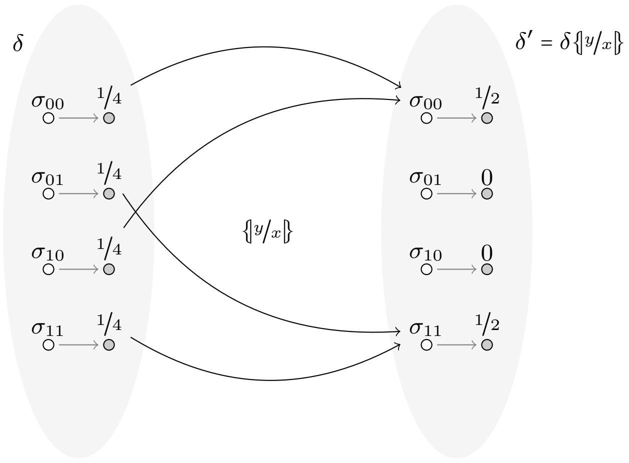

Fig. 4. The assignment $x \coloneqq y$ from an initial uniform distribution on the state space $\mathcal{S} = \{0,1\} \times \{0,1\}$

$$
\begin{array}{rcl}
\mathit{Abort} & \widehat{=} & \|\delta'\| \leq \|\delta\| \\
\mathit{Miracle} & \widehat{=} & (\delta = \epsilon) \wedge (\delta' = \epsilon) \\
\mathit{Skip} & \widehat{=} & \delta' = \delta \\
\underline{v} \coloneqq \underline{e} & \widehat{=} & \delta' = \delta\{\!|\underline{e}/\underline{v}|\!\} \\
A;B & \widehat{=} & \exists \delta_m \bullet A(\delta, \delta_m) \wedge B(\delta_m, \delta') \\
\mathit{choice}\big(A, B, \mathcal{X}\big) & \widehat{=} & \exists \pi, \delta_A, \delta_B \bullet \pi \in \mathcal{X} \wedge A(\delta\langle\pi\rangle, \delta_A) \wedge B(\delta\langle\bar{\pi}\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
c * A & \widehat{=} & \mathit{lfp}\, X \bullet \mathit{choice}\big((A;X), \mathit{Skip}, \{\iota\langle c\rangle\}\big)
\end{array}
$$

Fig. 5. UTP semantics for different programming constructs

The miraculous program $\mathit{Miracle}$ is defined as $(\delta = \epsilon) \wedge (\delta' = \epsilon)$: this is a different from the standard UTP theory, where it is simply false. This definition coincides with the standard one for most pairs of before- and after-distributions, with the exception of (ϵ, ϵ): this makes sure that $\mathit{Miracle}$ is a unit for nondeterministic choice.

Program Skip makes no changes and immediately terminates.

Assignment remaps the distribution as already discussed.

Sequential composition is characterised by the existence of a "mid-point" distribution that is the outcome of the first program, which is then fed into the second. It should be noted at this juncture that we are quantifying over function quantities, such as δ or π — this makes our logic at least second-order, even if the state spaces are finite (the range $[0,1]$ is not).

The choice operator takes a weighting distribution π, uses it with its complementary distribution $\bar{\pi} = \iota - \pi$) to weight the distributions resulting from the left- and right-hand side respectively, and existentially quantifies it over the set of distributions $\mathcal{X} \subseteq \mathcal{D}_w$, where $\mathcal{D}_w \subset \mathcal{D}$ is the set of all weighting distributions over the program state under consideration. We have termed this operator as the *generic choice* as it generalises the standard choice constructs:

- for $\mathcal{X} = \{\iota\langle c\rangle\}$ we have conditional choice:

$$\begin{aligned} A \lhd c \rhd B &= \mathit{choice}\big(A, B, \{\iota\langle c\rangle\}\big) \\ &= \exists \delta_A, \delta_B \bullet A(\delta\langle c\rangle, \delta_A) \land B(\delta\langle\neg c\rangle, \delta_B) \land \delta' = \delta_A + \delta_B \end{aligned}$$

- for $\mathcal{X} = \{p \cdot \iota\}$ we have probabilistic choice:

$$\begin{aligned} A \ {}_p\!\oplus B &= \mathit{choice}\big(A, B, \{p \cdot \iota\}\big) \\ &= \exists \delta_A, \delta_B \bullet A\big(p \cdot \delta, \delta_A\big) \land B\big((1-p) \cdot \delta, \delta_B\big) \land \delta' = \delta_A + \delta_B \end{aligned}$$

- for $\mathcal{X} = \mathcal{D}_w$ we have non-deterministic choice:

$$\begin{aligned} A \sqcap B &= \mathit{choice}\big(A, B, \mathcal{D}_w\big) \\ &= \exists \pi, \delta_A, \delta_B \bullet A(\delta\langle\pi\rangle, \delta_A) \land B(\delta\langle\bar{\pi}\rangle, \delta_B) \land \delta' = \delta_A + \delta_B \end{aligned}$$

The usual notations for conditional, probabilistic and non-deterministic choice will be used as syntactic sugar in the remainder of this document.

Program Abort is a zero for non-deterministic choice, whereas the program $\mathit{Miracle}$ is a unit.

Using the customary notation for conditional choice enlightens the definition of while-loops, which can be rewritten in a more familiar fashion as:

$$c \ast A \mathrel{\hat{=}} \mathit{lfp}\, X \bullet (A ; X) \lhd c \rhd \mathit{Skip}$$

They are characterized as fixpoints of the appropriate functional, with respect to the ordering defined by the refinement relation, details of which can be found in [MM04, BB11] and are beyond the scope of this paper.

Healthiness Conditions. The distributional framework is characterised by the following healthiness conditions:

Dist1: the *feasibility* condition assures that the probability of termination cannot be greater than that of having started:

$$\|\delta'\| \le \|\delta\|$$

Dist2: the *monotonicity* condition states that increasing δ implies that the resulting δ' increases as well:

$$\mathcal{P}(\delta_1, \delta_1') \land \mathcal{P}(\delta_2, \delta_2') \land \delta_2 > \delta_1 \Rightarrow \delta_2' \ge \delta_1'$$

Dist3: the *scaling* condition is about multiplication by a (not too large and non-negative[3]) constant, which distributes through commands:

$$\forall a \in \mathbb{R}^+ \land \|a \cdot \delta\| \le 1 \bullet \mathcal{P}(\delta, \delta') \Leftrightarrow \mathcal{P}(a \cdot \delta, a \cdot \delta')$$

[3] Mathematically the relation holds also if this is not met, but in that case the distribution $a \cdot \delta$ may not be a probability distribution.

Dist4: the *convexity* condition poses restrictions on the space of possible program images[4], which is strictly a subset of $\wp\mathcal{D}$, the powerset of $\mathcal{D}$:

$$(\mathcal{P}_1 \sqcap \mathcal{P}_2)(\delta, \delta') \Rightarrow \delta' \geq \min\big(\mathcal{P}_1(\delta) \cup \mathcal{P}_2(\delta)\big)$$

Here $\mathcal{P}_i(\delta)$ denotes the set of all δ' that satisfy $\mathcal{P}_i(\delta, \delta')$.

We refer to this set as to the *program image* of $\mathcal{P}_i$ — we will use this concept to show the program lattice in the case of designs in Figure 6.

3.2 A Probabilistic Theory of Designs

We have used the framework above to give semantics to a probabilistic theory of designs [BB12a].

A big difference from the standard theory is that we did not need to use the auxiliary variables ok and ok': in fact the variable δ records implicitly if the program has started, as for each state it gives a precise probability that the program is in that initial state, while the variable δ' records implicitly if the program has finished, as for each state it gives a precise probability that the program is in that final state.

We can therefore relate the fact that a program has started with probability 1 with the fact that δ is a full distribution (*i.e.* $\|\delta\| = 1$): in other words the statement $ok = true$ can be translated to the statement $\|\delta\| = 1$.

Conversely a program for which $\delta = \epsilon$ is a program that has not started. Obviously there are all situations in between, where the fact of δ being a sub-distribution accounts for the program having started with probability $\|\delta\| < 1$.

Similarly if δ' is a full distribution, then the program terminates with probability 1: coherently we can translate the statement $ok' = true$ to the statement $\|\delta'\| = 1$. In general the weight of δ' is the probability of termination: if the program reaches an after-distribution whose weight is strictly less than 1, then termination is not guaranteed (and in particular if $\delta' = \epsilon$ it is certain that it will not terminate).

With these considerations in mind, it is straightforward, given a standard design $\mathcal{P}re \vdash \mathcal{P}ost$, to derive the corresponding probabilistic design:

$$\mathcal{P}re \vdash \mathcal{P}ost \equiv \|\delta\langle\mathcal{P}re\rangle\| = 1 \Rightarrow \|\delta'\langle\mathcal{P}ost\rangle\| = 1$$

This expression tells us that we have a valid design if whenever the before-distribution δ is a full distribution which is null everywhere $\mathcal{P}re$ is not satisfied (and therefore $\delta = \delta\langle\mathcal{P}re\rangle$), then the resulting after-distribution δ' is a full distribution which is null everywhere $\mathcal{P}ost$ is not satisfied (and therefore $\delta' = \delta'\langle\mathcal{P}ost\rangle$).

In other words both δ and δ' belong to the set $\mathcal{D}_p \cap \mathcal{B}(\epsilon, 1)$, which we note as $\partial\mathcal{D}_p$ (with a bit of notation abuse), where $\mathcal{D}_p \subset \mathcal{D}$ is the set of all probability distributions and $\mathcal{B}(\epsilon, 1)$ is the closed unitary ball[5] centered on the empty distribution ϵ.

[4] This is a consequence of the purely random non-deterministic model adopted in the distributional framework, yielding a result analogous to the set $\mathbb{H}S$ from [MM04].

[5] The norm of δ is $\|\delta\|$, and the distance function of the space is $d(\delta_1, \delta_2) \triangleq \|\delta_2 - \delta_1\|$.

In a similar way we can find a probabilistic version of other standard designs:

- assignment requires the right-hand expression to be defined everywhere in the state space, otherwise it reduces to false:

$$\underline{v} := \underline{e} \;\widehat{=}\; \|\delta\| = 1 \Rightarrow \|\delta'\| = 1 \wedge \delta' = \delta\{\!|\underline{e}/\underline{v}|\!\}$$

- the Skip construct preserves the before-distribution unchanged:

$$\mathit{Skip} \;\widehat{=}\; \|\delta\| = 1 \Rightarrow \delta' = \delta$$

- probabilistic designs form a lattice as well (with respect to the ordering induced by the $\Rightarrow$ relation). The bottom of the lattice is Abort, which is again true as in the standard theory:

$$\mathit{Abort} \;\widehat{=}\; \mathit{true}$$

- Chaos is a program that guarantees termination, but in an unspecified state[6]:

$$\mathit{Chaos} \;\widehat{=}\; \|\delta\| = 1 \Rightarrow \|\delta'\| = 1$$

- the top of the lattice is $\mathit{Miracle}$:

$$\mathit{Miracle} \;\widehat{=}\; \|\delta\| < 1$$

These new definitions preserve the validity of the healthiness conditions H1–H4, as on the other hand do all the constructs from the distributional framework [BB12a]: for this reason we can think of a variation and relax the constraints on the weights of the before- and after-distributions — so we use the programming constructs in Figure 5 exactly with the semantics presented there. By doing so we can fully exploit the potential of the distributional framework towards modelling situations where the probability of having started is less than 1: with a small modification we can recast the notion of total correctness by restricting Dist1 to a variant Dist1-TC (which implies Dist1), stating that:

$$\|\delta\| = \|\delta'\|$$

This requires a program to terminate with the same probability p with which it has started:

$$\|\delta\| = p \wedge \mathit{Pre} \Rightarrow \|\delta'\| = p \wedge \mathit{Post}$$

The role of preconditions and postconditions is that of restricting the range of acceptable before- and after-distributions (and therefore act as restrictions to be applied to δ and δ' respectively) — this allows us to express desirable characteristics of a program in great detail.

Through our distributional framework we therefore obtain a richer theory where corresponding healthiness conditions hold, even without the introduction of the auxiliary variables $\mathit{ok}, \mathit{ok}'$ — the link with the standard model is discussed in [BB12a]. Moreover the use of distributions enables us to evaluate the probability both of termination and of meeting a set of arbitrary postconditions as a function of the initial distribution (which determines the probability of meeting any required precondition).

[6] In other words $\mathit{Chaos} \equiv \mathit{true} \vdash \mathit{Abort}_R$, where the subscript R indicates that we are talking of the relational version of Abort, from Figure 5.

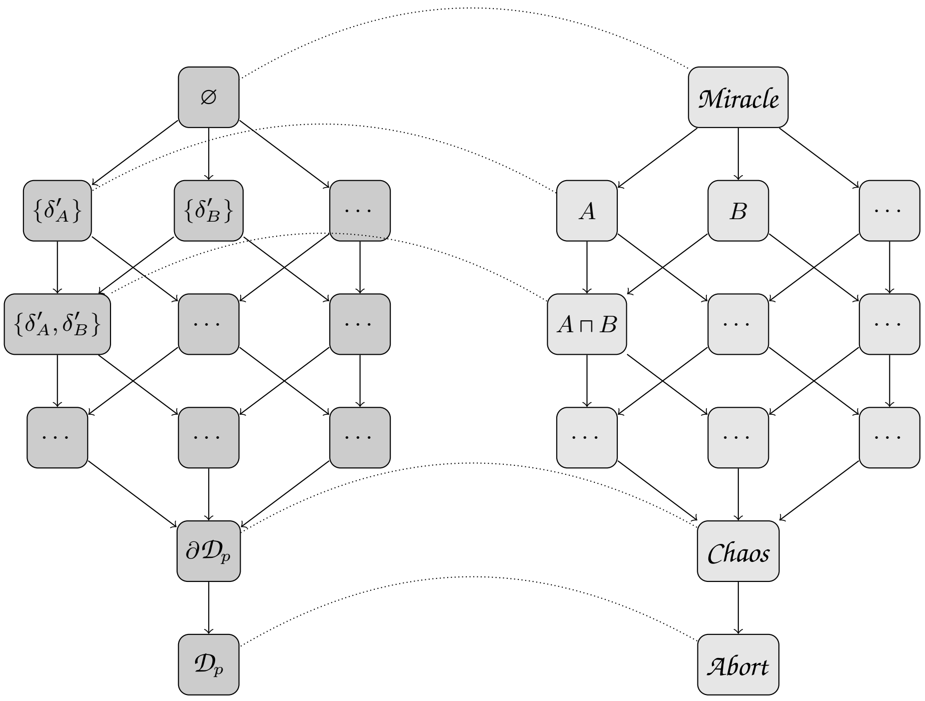

Fig. 6. Program image lattice ($\subseteq$ relation) and program lattice ($\Rightarrow$ relation) for probabilistic designs

4 Probabilistic CSP, UTP-Style

We have seen that the UTP theory of CSP is built on that of designs, with the introduction of three other pairs of auxiliary variables, notably $\mathit{wait}, \mathit{tr}, \mathit{ref}$ and their dashed counterparts.

We recall their roles in the theory:

– $\mathit{wait}, \mathit{wait}'$ are boolean variables recording if the program is waiting for interaction with the environment;
– $\mathit{tr}, \mathit{tr}'$ record the list of events happened during the program run;
– $\mathit{ref}, \mathit{ref}'$ are sets containing the event refused by the program.

They are in addition to $\mathit{ok}, \mathit{ok}'$, already added when going from the relational theory towards the concept of designs: the distributional framework spared us from having to add these variables when creating the concept of probabilistic designs, as we do not need to use them — we have in fact argued that this information is contained implicitly in the distributions δ, δ', as their weight corresponds exactly to the probability that a particular program step has started or finished, respectively.

Information about divergent states remains implicit in the distributions: the probability of being in such a situation is precisely $(1 - \|\delta'\|)$.

In some sense the "ok" part of a distribution is mapped to the support of δ', whereas the "not-ok" part gets disregarded.

We can therefore build on the theory of probabilistic designs presented in §3.2 to get to a probabilistic theory of CSP only by adding the remaining three pairs of auxiliary variables.

Their meaning will be the same as in the standard theory. The question is: what is the best way to embed them in the probabilistic theory of designs? We may be tempted to introduce them as auxiliary variables alongside with the program distribution, but the same reasons that were brought up to decide in favour of an approach that lumps all of the variables together into a single composite observation variable, require us to work on states with the following shape:

$$\sigma : (\underline{v}, wait, tr, ref) \to \underline{\mathcal{W}} \times \mathbb{B} \times Event\text{-seq} \times Event\text{-set}\,,$$

where $\mathcal{W}$ is the set of possible values for the program variables.

This allows us to embed all of the remaining auxiliary variables in the state domain, and therefore this simplifies the definitions of the different programming constructs and healthiness conditions, compared to the traditional reactive definitions that use $ok, wait, tr, ref$ as auxiliary variables — this is a novel approach.

4.1 R1

For example let us take the traditional R1, which states:

$$P = P \land (tr \le tr')$$

In a probabilistic world this must hold pointwise for each couple of states (σ, σ') from the before- and after-distributions that are related by the program.

If we write this in the case of a single state σ (*i.e.* we take a point distribution η_σ as the before-distribution), the trace in the before-state σ must be a prefix of the trace in all of the possible after-states σ' from the support[7] of the resulting after-distribution δ'.

This must hold true for all states in the state space, so the formulation of the probabilistic R1 is:

$$P(\delta,\delta') \;=\; P(\delta,\delta') \;\land\; \Big(\forall \sigma \bullet P(\eta_\sigma,\delta') \implies \big(\forall \sigma' \in \mathrm{supp}(\delta') \bullet \sigma(tr) \le \sigma'(tr)\big)\Big)$$

where we have used the functional notation $\sigma(tr)$ to stand for the evaluation of tr on σ.

From this formulation we can clearly see that divergent states do not take part in the verification of the condition R1; in addition, it is worth pointing out that, according to this definition, a totally divergent program (which yields $\delta' = \epsilon$ for any initial δ) is R1-healthy.

[7] The support of a function is the subset of its domain where the function is non-null.

4.2 R2

Healthiness condition R2 states that the initial value of tr cannot have any influence on the evolution of the program, which determines only the tail $(\mathit{tr}' - \mathit{tr})$:

$$P(\mathit{tr}, \mathit{tr}') = \exists s \bullet P(s, s \frown (\mathit{tr}' - \mathit{tr}))$$

As we did above we first look at the case of point distributions, where a possible formulation is the following:

$$P(\eta_\sigma, \delta') = \exists s \bullet P(\eta_\sigma\{\!|{}^{s}/_{\mathit{tr}}|\!\}, \delta'\{\!|{}^{s\frown(\mathit{tr}-\sigma(\mathit{tr}))}/_{\mathit{tr}}|\!\})$$

Here we have used the remap operator to "change" the value of the trace in the spirit of R2 over all states.

This gives a sort of "substitution rule" that allows us to replace a state σ with another state ζ that differs only for the value of tr in the before-distribution, whereas in the after-distribution a part δ'_σ (accounting for the contribution of σ) is replaced by a new part δ'_ζ (accounting for the contribution of ζ):

$$P(\delta, \delta') = \forall\sigma\exists s \bullet (\zeta = \sigma\{{}^{s}/_{\mathit{tr}}\}) \wedge P\big((\delta - \delta_\sigma + \delta_\zeta), (\delta' - \delta'_\sigma + \delta'_\zeta)\big)$$

where δ_σ and δ_ζ are point distributions scaled down by the probability of σ, *i.e.* $\delta_\sigma = \delta(\sigma) \cdot \eta_\sigma$ and $\delta_\zeta = \delta(\sigma) \cdot \eta_\zeta$.

4.3 R3

Before getting to R3 we have to define the probabilistic version of the reactive Skip, denoted $\mathit{I\!I}$.

According to the standard theory of reactive designs [HH98], $\mathit{I\!I}$ is defined as:

$$\mathit{I\!I} \ \widehat{=}\ (\neg \mathit{ok} \wedge \mathit{tr} \leq \mathit{tr}') \ \vee \ (\mathit{ok}' \wedge \mathit{tr}' = \mathit{tr} \wedge \mathit{wait}' = \mathit{wait} \wedge \mathit{ref}' = \mathit{ref})$$

This definition has to distinguish the case of divergence (when it does not enforce anything other than trace elongation) from the case of non-divergence (when it states that all variables are left unchanged), and as a result it is much more complicated than the pure relational skip which is simply:

$$\underline{v}' = \underline{v}$$

The choice of embedding the auxiliary variables in the state function σ (and having left all information about divergence implicit in δ, δ') starts to pay out here, as it enables us to keep such an easy definition as well:

$$\mathit{I\!I} \ \widehat{=}\ \delta' = \delta$$

In other words all non-divergent states are preserved as they are, whereas now there is no statement on divergent states — other than the implicit one that the overall probability of divergence must be left unchanged.

R3 does not mention tr, tr':

$$P = I\!I \lhd wait \rhd P$$

As a result this is pretty straightforward to express in a probabilistic setting, as we can use directly the semantics of the conditional construct presented in §3.1:

$$\begin{array}{ll}
& I\!I \lhd wait \rhd P \\
\equiv & \quad \text{definition of conditional} \\
& \exists \delta_A, \delta_B \bullet I\!I(\delta\langle wait\rangle, \delta_A) \;\wedge\; P(\delta\langle \neg wait\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
\equiv & \quad \text{definition of } I\!I \\
& \exists \delta_A, \delta_B \bullet I\!I(\delta\langle wait\rangle, \delta_A) \wedge \delta_A = \delta\langle wait\rangle \;\wedge\; P(\delta\langle \neg wait\rangle, \delta_B) \wedge \delta' = \delta_A + \delta_B \\
\equiv & \quad \text{one-point rule on } \delta_A \\
& \exists \delta_B \bullet I\!I(\delta\langle wait\rangle, \delta\langle wait\rangle) \;\wedge\; P(\delta\langle \neg wait\rangle, \delta_B) \wedge \delta_B = \delta' - \delta\langle wait\rangle \\
\equiv & \quad \text{one-point rule on } \delta_B \\
& I\!I(\delta\langle wait\rangle, \delta\langle wait\rangle) \;\wedge\; P(\delta\langle \neg wait\rangle, \delta' - \delta\langle wait\rangle)
\end{array}$$

And therefore.

$$P(\delta, \delta') = I\!I(\delta\langle wait\rangle, \delta\langle wait\rangle) \;\wedge\; P(\delta\langle \neg wait\rangle, \delta' - \delta\langle wait\rangle)$$

We split the before-distribution into two parts, one where $wait$ is true and that equals the corresponding after-distribution, and one where it is not and that has evolved into the difference of the total after-distribution δ' and the part $\delta\langle wait\rangle$ that did not evolve.

This can be simplified down to:

$$P(\delta, \delta') = P(\delta\langle \neg wait\rangle, \delta' - \delta\langle wait\rangle)\,.$$

4.4 CSP1 and CSP2

At this stage the readers with prior knowledge of *CSP* in the *UTP* framework may be surprised that the end of this paper is approaching and yet we have not mentioned two other healthiness conditions, namely CSP1 and CSP2.

The reason for our omission is that another advantage of the distributional framework is that compliance with these healthiness conditions is subsumed by other conditions, as we are now going to show.

In standard *CSP*, CSP1 states that:

$$P = P \vee (\neg ok \wedge tr \le tr')$$

As all information about divergent states is kept implicit in distributions, we can argue that this healthiness condition is stripped down to the identity $P = P$.

In some sense, all states which are "ok" evolve from the support of the before-distribution towards a state in the support of the after-distribution, which is "ok′", or diverge to a state, which is "not-ok′" and is not part of the support of the after-distribution, effectively getting out of the game; on the other hand all states which are "not-ok" are not part of the support of the before-distribution and have no means to get back in the game.

Probabilistic reactive programs are therefore CSP1-healthy by design, as $P(\delta, \delta')$ already states that either a state evolves according to what is described by δ, δ' or diverges.

Our formalism does not allow us to express the trace-elongation property for divergent states, but after all it is not crucial information — they diverge, that's already bad enough!

The other healthiness condition, CSP2, states that:

$$P; J = P$$

where

$$J \mathrel{\widehat{=}} \underline{v}' = \underline{v} \wedge (ok \Rightarrow ok') \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref$$

In the probabilistic world based on distribution this reduces to:

$$P; II = P$$

which is nothing but H3. In fact:

$$\begin{aligned} J &\mathrel{\widehat{=}} \left(\underline{v}' = \underline{v} \wedge (ok \Rightarrow ok') \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref\right) \\ &\equiv \left(\underline{v}' = \underline{v} \wedge ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref\right) \vee \\ &\qquad \vee \left(\underline{v}' = \underline{v} \wedge \neg ok \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref\right) \\ &\equiv II \vee \left(\underline{v}' = \underline{v} \wedge \neg ok \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref\right) \end{aligned}$$

And again the part with $\neg ok$ gets disregarded, thus the reactive program J in the probabilistic world coincides with II — and there we have that CSP2 collapses to H3.

5 Conclusion

We have built a framework using the notion of distributions on the state space: through distributions we are able to associate a probability with each state.

If we use predicates stating relations among distributions we can build a *UTP* theory of programs that naturally embeds probability, so that probabilistic choice and non-deterministic choice can co-exist in the same framework.

We have extended this theory first to generalise the standard *UTP* theory of designs, and then we have built on that a theory of reactive programs.

The peculiarity of this approach is that divergent states are implicitly accounted for by sub-distributions, where the weight is strictly less than one: a divergent state does not belong to the domain of a distribution (in some sense all states which are "not-ok" are disregarded), and the overall probability of being in a divergent state is equal to the difference between 1 and the distribution weight.

Probabilistic versions of healthiness conditions R1, R2 and R3 hold in the probabilistic theory, whereas healthiness conditions CSP1 and CSP2 are subsumed by the framework.

References

[BB11] Bresciani, R., Butterfield, A.: Towards a UTP-style framework to deal with probabilities. Technical Report TCD-CS-2011-09, FMG, Trinity College Dublin, Ireland (August 2011)

[BB12a] Bresciani, R., Butterfield, A.: A probabilistic theory of designs based on distributions. In: Wolff, B., Gaudel, M.-C., Feliachi, A. (eds.) UTP 2012. LNCS, vol. 7681, pp. 105–123. Springer, Heidelberg (2013)

[BB12b] Bresciani, R., Butterfield, A.: A UTP semantics of pGCL as a homogeneous relation. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 191–205. Springer, Heidelberg (2012)

[But10] Butterfield, A. (ed.): UTP 2008. LNCS, vol. 5713. Springer, Heidelberg (2010)

[CS09] Chen, Y., Sanders, J.W.: Unifying probability with nondeterminism. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 467–482. Springer, Heidelberg (2009)

[Dij76] Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)

[DS06] Dunne, S., Stoddart, B. (eds.): UTP 2006. LNCS, vol. 4010. Springer, Heidelberg (2006)

[He10] He, J.: A probabilistic BPEL-like language. In: Qin [Qin 2010], pp. 74–100 (2010)

[Heh84] Hehner, E.C.R.: Predicative programming — Part I & II. Commun. ACM 27(2), 134–151 (1984)

[HH98] Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)

[Hoa85] Hoare, C.A.R.: Programs are predicates. In: Proceedings of a discussion meeting of the Royal Society of London on Mathematical Logic and Programming Languages, Upper Saddle River, NJ, USA, pp. 141–155. Prentice-Hall (1985)

[HS06] He, J., Sanders, J.W.: Unifying probability. In: Dunne and Stoddart [DSO 2006], pp. 173–199 (2006)

[Koz81] Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. 22(3), 328–350 (1981)

[Koz85] Kozen, D.: A probabilistic PDL. J. Comput. Syst. Sci. 30(2), 162–178 (1985)

[Mis00] Mislove, M.W.: Nondeterminism and probabilistic choice: Obeying the laws. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 350–365. Springer, Heidelberg (2000)

[MM04] McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science. Springer (2004)

[MMSS96] Morgan, C., McIver, A., Seidel, K., Sanders, J.W.: Refinement-oriented probability for CSP. Formal Asp. Comput. 8(6), 617–647 (1996)

[Mor04] Morgan, C.: Of probabilistic *Wp* and *CSP*—and compositionality. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) CSP 2004. LNCS, vol. 3525, pp. 220–241. Springer, Heidelberg (2005)

[OCW09] Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Asp. Comput 21(1-2), 3–32 (2009)

[Qin10] Qin, S. (ed.): UTP 2010. LNCS, vol. 6445. Springer, Heidelberg (2010)