UTP Semantics for Shared-State, Concurrent, Context-Sensitive Process Models

Andrew Butterfield
Lero @ TCD
School of Computer Science and Statistics
Trinity College, Dublin 2, Ireland
Email: Andrew.Butterfield@lero.ie

Anila Mjeda
Lero @ TCD
School of Computer Science and Statistics
Trinity College, Dublin 2, Ireland
Email: Anila.Mjeda@lero.ie

John Noll
Lero @ UL
Tierney Building
University of Limerick
Email: John.Noll@lero.ie

***Abstract*—Process Modelling Language (PML) is a notation for describing software development and business processes. It takes the form of a shared-state concurrent imperative language describing tasks as activities that require resources to start and provide resources when they complete. Its syntax covers sequential composition, parallelism, iteration and choice, but without explicit iteration and choice conditions. It is intended to support a range of context-sensitive interpretations, from a rough guide for intended behaviour, to being very prescriptive about the order in which tasks must occur. We are using Unifying Theories of Programming (UTP) to model this range of semantic interpretations, with formal links between them, typically of the nature of a refinement. We address a number of challenges that arise when trying to develop a compositional semantics for PML and its shared-state concurrent underpinnings, most notably in how UTP observations need to distinguish between dynamic state-changes and static context parameters. The formal semantics are intended as the basis for tool support for process analysis, with applications in the healthcare domain, covering such areas as healthcare pathways and software development and certification processes for medical device software.** **http://doi.ieeecomputersociety.org/10.1109/TASE.2016.22**

I. Introduction

Programming-like notations have been used to describe business processes and workflows for many years [1]–[3]. There is considerable interest at present in healthcare systems in so-called clinical pathways, that describe processes for managing patient care. These too can be described using general business process notations [4]–[6]. Deploying process models in the medical domain in practise requires flexible interpretations of those models [2], [4], [6]–[9] .

PML is such a language [10], developed originally for modelling software development processes, but applicable to a much wider range of activities, including clinical pathways [11]. It is designed to encourage a flexible approach to its interpretation and deployment. This is most obvious when one considers that the condition and iterative constructs of the language have no condition predicates, relying on the judgement of those enacting the process to determine which conditional branch should be taken, or when a loop should terminate.

In this paper we present results obtained while developing a range of formal semantics for PML using the Unifying Theorems of Programming (UTP) framework [12] to support reasoning about flexible deployment. We define a UTP semantics for shared global state concurrency, as a way to get a suitable semantics for strict and flexible PML, inspired by the work of Woodcock and Hughes on unifying parallel programming (UTPP, [13]). We present both a formal semantics for a "weak" interpretation of PML, as well as a unified theory of concurrent programs (UTCP) that will provide a baseline theory for modelling more "flexible" and "strong" interpretations of PML. Part of our contribution is extending the UTP methodology to use non-homogenous relations that mix dynamic state-change (observations with before- and after-values) along with static context parameters (observations whose value is unchanged during program execution). We also develop a notion of label generators to allow us to describe flow of control in a compositional manner.

The structure of this paper is as follows: In §II we give a quick introduction to an abstract form of PML, while in §III we provide a quick overview of UTP. We then move on to present the weak semantics for PML in §IV, the baseline UTCP semantics in §V, and then to relate the two in §V-H, where we also discuss future work. We then describe related work (§VI) and conclude (§VII).

II. Process Modelling Language

Process Modelling Language (PML) [10] is a shared-state concurrent imperative language describing named basic actions ($N?rr!pr$) as activities that require resources to start (rr) and provide resources when they complete (pr). Actions are non-destructive in that the required resources are not consumed but remain in place. Its concrete syntax is C-like, but we present a simpler abstract syntax here:

$$
\begin{array}{rcl}
N \in Name & & \\
A \in Action & ::= & N?rr!pr \\
P,Q \in PML & ::= & A \mid P \mathbin{;;} Q \mid P \lhd\rhd Q \mid P \parallel Q \mid P^{\omega}
\end{array}
$$

We use $P \mathbin{;;} Q$ and $P \parallel Q$ to denote respectively sequential and parallel composition. Both the conditional ($P \lhd\rhd Q$) and the loop construct P^{ω} are unusual in that they have no explicit conditions. Instead the decision of which branch to take or whether or not to end the loop is left unspecified—this is one aspect of the flexible nature of the language. In its original intended use, it was left to the people enacting the business

process described by PML to use their judgement to determine which branch to take or when a process has been repeated enough.

We are developing a formal PML semantics framework which can investigate and relate at least three interpretations of a PML description:

- **Strict:** the behaviour follows strictly according to the control-flow structure, becoming deadlocked if control requires execution of actions whose required resources are not available.
- **Flexible:** the behaviour is guided by the control flow, but actions can run out of sequence if their required resources become available before the control flow has determined that they should start.
- **Weak:** control flow is completely ignored, and execution simply iterates the non-deterministic choice of actions whose resources are available (also known as the "dataflow interpretation").

In our framework, the three interpretations form a refinement hierarchy:

$$Weak \sqsubseteq Flexible \sqsubseteq Strict$$

As an example, consider the following PML description:

$$A?!r_1 \;;;\; (B?r1!r_2 \parallel C?r_2!r_3) \;;;\; D?r_1!r_4$$

In the strict interpretation, only one sequence is possible, namely A, B, C, D because (i) A is required by control-flow to precede both B and C, and (ii) D is forced to be last. In the weak interpretation, we ignore the flow control, and are left with resource constraints only (B must follow A, C must follow B and D must follow$A$) This admits three possible enactment sequences: A, B, C, D, or A, D, B, C, or A, B, D, C.

In this paper we focus on the weak semantics, and on developments we have done within the Unifying Theories of Programming (UTP) framework [12] to create a semantic baseline on top of which both the flexible and strict semantics can be constructed.

III. Unifying Theories of Programming

The Unifying Theories of Programming framework [12] uses predicate calculus to define before-after relationships over appropriate collections of free observation variables. The before-variables are undashed, while after-variables have dashes[1]. Some observations correspond to the values of program variables (v, v'), while others deal with important observations one might make of a running program, such as start (ok) and termination (ok') or event traces (tr, tr'). The set of observation variables associated with a theory or predicate is known as its *alphabet*. For example, the meaning of an assignment statement might be given as follows:

$$x := e \quad \widehat{=} \quad ok \implies ok' \wedge x' = e \wedge \nu' = \nu$$

The definition says that once started, the assignment terminates, with the final value of variable x set equal to the value of expression e in the before-state, while the other variables, denoted collectively by ν, remain unchanged. In UTP, a predicate of the form $ok \wedge P \implies ok' \wedge Q$ is called a *Design*, denoting a total-correctness assertion, is typically shortened to $P \vdash Q$. UTP supports specification languages as well as programming ones, and a general notion of refinement as universally-closed reverse implication:

$$S \sqsubseteq P \mathrel{\widehat{=}} [P \implies S]$$

A key part of the UTP framework is the use of healthiness conditions to ensure that the predicates actually correspond to feasible program outcomes. These are typically defined using idempotent and monotonic predicate-transformers ($\mathbf{H}$) that make an un-healthy predicate healthy ($\mathbf{H}(Bad) = Better$), while leaving healthy ones untouched ($\mathbf{H}(Good) = Good$). For example, the unhealthy predicate $\neg ok \wedge ok'$ (I've finished even though I didn't start!) is ruled out by the following predicate-transformer:

$$\mathbf{H1}(P) = \neg ok \vee P$$

In contrast, applying $\mathbf{H1}$ to the definition of assignment leaves it unchanged. Most of the healthiness conditions define sets of healthy predicates that form complete lattices, where the bottom element is the most liberal specification ("chaos"), and the top element is the infeasible program that satisfies any specification ("miracle"). The healthiness conditions that do not produce lattices are typically optional ones that can be invoked to rule out miracles, and which effectively chop off the upper infeasible predicates, and leaving fully refined programs as maximal elements.

IV. Weak PML

We give the semantics of the weakest interpretation of PML, where we only consider the dataflow-like behaviour induced by the "requires"(?) and "provides" (!) annotations on basic actions. We shall assume a simple model of resources, as uninterpreted entities that are distinguishable, and where the only property of interest is whether or not any given resource is present in the system. What we want to observe is how the resources in the system evolve over time, as well as keeping track (by name) of which basic actions were executed. We introduce three observations, all of which can be made both before and after a PML run: starting, termination ($ok, ok' : \mathbb{B}$); resources in the system ($r, r' : \mathcal{P}\, Res$); and basic actions executed ($h, h' : Name^*$).

Due to space limitations, we do not give a detailed account of the healthiness conditions here, but note that we have the usual Design conditions ($\mathbf{H1}, \mathbf{H2}, \mathbf{H3}$) as well as modified versions of some of the Reactive systems conditions [12, Chp 8] which we present here without comment:

$$\begin{aligned} \mathbf{P1}(P) &\quad \widehat{=} \quad P \wedge h \leq h' \\ \mathbf{P2}(P) &\quad \widehat{=} \quad \exists h_0 \bullet P[h_0, h_0 \frown (h' - h)/h, h'] \end{aligned}$$

In particular, we have a complete lattice with bottom $\mathbf{true}$ and top $\neg ok$.

[1]We follow [12, §1,p25] in using 'dashed', rather than 'primed'

Our semantics has two parts, an action collection phase ($Coll$) and a dynamic execution model ($doStep$, $RunW$). The collection function works through a PML description and collects all the basic actions, recording them in a finite map. This works because of the PML requirement that every textual occurrence of a basic action has a unique name or identifier.

$$\begin{array}{rcl} Coll & : & PML \to (Name \pfun Action) \\ Coll(N?rr!pr) & \mathrel{\widehat=} & \{N \mapsto (N?rr!pr)\} \\ Coll(P \oplus Q) & \mathrel{\widehat=} & Coll(P) \uplus Coll(Q), \quad \oplus \in \{;, \lhd\rhd, \|\} \\ Coll(P^\omega) & \mathrel{\widehat=} & Coll(P) \end{array}$$

Here $\uplus$ denotes map disjoint union. The meaning of a basic action is a design that runs when its required resources are available, here defined with the notion of a guarded action ($g\&A$):

$$\begin{array}{rcl} N?rr!pr & \mathrel{\widehat=} & \mathbf{true} \vdash rr \subseteq r \;\&\; r' = r \cup pr \wedge h' = h \frown \langle N \rangle \\ g \;\&\; A & \mathrel{\widehat=} & A \lhd g \rhd \neg ok \end{array}$$

Given a basic-actions map ϖ, we can determine, w.r.t to the current resource set r, if there is an enabled action whose execution will extend that resource set ($canGo$):

$$\begin{array}{rcl} isEna(N?rr!pr) & \mathrel{\widehat=} & rr \subseteq r \\ isDone(N?rr!pr) & \mathrel{\widehat=} & pr \subseteq r \\ canGo(\varpi) & \mathrel{\widehat=} & \exists N \bullet N \in \dom \varpi \\ & & \quad \wedge\, isEna(\varpi N) \wedge \neg isDone(\varpi N) \end{array}$$

We describe a single step of the system as one that makes a non-deterministic choice out of all of the currently enabled actions and executes it:

$$doStep(\varpi) \quad \mathrel{\widehat=} \quad \exists N \bullet N \in \dom \varpi \wedge isEna(\varpi N) \wedge \varpi N$$

Here we exploit a standard part of the UTP methodology ("programs are predicates") that a action ($N?rr!pr$ or ϖN) is considered the same thing as its defining predicate.

We run a PML description ($RunW$) by repeatedly doing a step so long as it is possible to make progress (provide more resources):

$$RunW(\varpi) \quad \mathrel{\widehat=} \quad canGo(\varpi) * doStep(\varpi)$$

Here $c * P$ is UTP notation for **while** c **do** P.

The dynamic semantics of a PML description P is therefore given as

$$P \quad = \quad RunW(Coll(P))$$

The following distributive law is an easy result, for $\oplus$ ranging over $;, \lhd\rhd, \|$:

$$\begin{array}{rcl} RunW(Coll(P \oplus Q)) & = & RunW(Coll(P)) \\ & & \quad \vee\, RunW(Coll(Q)) \end{array}$$

We get one step law for Weak PML: picking one of the available ready actions, running it and continuing execution constitutes a refinement of that program. This refinement is all of the behaviours that are possible after that first (possibly non-deterministic) choice is made.

$$\begin{array}{c} canGo(\varpi) \wedge N \in \dom \varpi \wedge isEna(\varpi N) \\ \Longrightarrow \quad RunW(\varpi) \sqsubseteq (\varpi N); RunW(\varpi) \end{array}$$

V. Unifying Concurrency

The Flexible and Strict PML semantics have in common that they consider PML descriptions as denoting concurrent behaviour in the presence of global shared state (resources). They differ in the precise criterion used to determine when a particular task may actually run. Our semantics is inspired closely by that of UTPP [13], which translates a parallel program into an action system [14], whose semantics is given using UTP. An action is viewed as a behaviour coupled with a guard that establishes the conditions on observable state when that action may begin to execute. An action system is a collection of such guarded actions, that continually runs, with a non-deterministic choice being made whenever multiple guards are enabled at the same time [15]. In terms of UTP's lattice-theoretic approach, the system is a distributed join (non-deterministic choice) of all its basic actions, where a disabled (false) guard behaves the like the join unit value (top/miracle). Or, in simpler predicate language, a disjunction of tasks as Designs, with $\neg ok$ as the unit.

Our semantics steps back a slight bit from the action-system formulation, in that guards are de-emphasised, at least syntactically, but we still have the distributed join, with disabled actions as unit. We did this to reduce some of the mechanism associated with Designs, so as to make the nature of the parallel semantics easier to see. However, we expect our semantics can easily be "poured" back into a full action system framework without any difficulty.

Our key motivation in stepping away from action systems, and the UTPP formulation was to explore if we could eliminate one of the sources of non-compositionality in that semantics, namely that, while each basic statement had a starting label, its semantics was defined in terms of that label, and the labels of *whatever statements came after it*. These labels are analogous to the notion of an abstract program counter that is found in many approaches to reasoning about concurrent programs [16], so that it is possible to talk about the particular program statement that any particular thread is trying to execute. They allow the semantics to track the progress of each thread while also allowing the non-deterministic interleaving of atomic actions.

We wanted to get better compositionality, so that the semantics of any construct was independent of its context, and the semantics of any composite could be constructed from the semantics of its components. In the case of UTPP, it is not clear how much we have gained in terms of calculational ease, but the UTP paradigm we have developed in order to achieve this looks like it might be a framework for addressing a wider range of semantics where compositionality is difficult.

A. UTCP Observables

The key idea is that we use some observations to, in effect, act as abstractions of the surrounding (local) context. As actual context is built up by the use of language composites, those observations are then instantiated in such a way that either hides them as internal, or lets them play an appropriate role at the higher level.

The theory we built uses predicate variables to record observations of program behaviour in two distinct ways:

1) Making observations of dynamic state change, using un-decorated variables to record before-values, and dashed variables to denote the corresponding after-value, as is the norm in most UTP theories. We shall refer to these as *dynamic observables*. In our theory we shall use s to denote (shared) state and ls to denote the current set of active labels.

$$\begin{array}{rcl} s, s' & : & State \\ ls, ls' & : & \mathcal{P} Label \end{array}$$

2) Some observations record context information that is propagated throughout a program. This information does not change during the lifetime of a program execution, and so only needs to be recorded using un-dashed variables. We shall call these *static parameters*. In this theory we associate three static parameters with every construct: input and output labels (in,out) and a label generator (g).

$$\begin{array}{rcl} in, out & : & Label \\ g & : & Gen \end{array}$$

The alphabet we use for a UTCP program P is

$$\alpha P = \{s, s', ls, ls', g, in, out\}$$

However we will also talk about basic atomic actions A that only affect the global state:

$$\alpha A = \{s, s'\}$$

B. Basic UTP Building Blocks

The definition of the semantics of the UTCP language constructs, and of run, make use of the (almost) standard notions of skip, sequential composition and iteration in UTP. The versions used here are slightly non-standard because we have non-homogeneous relations, where the static parameters have no dashed counterparts. In essence we ignore the static parameters as far as flow-of-control is concerned:

$$\begin{array}{rcl} II & \widehat{=} & s' = s \wedge ls' = ls \\ P; Q & \widehat{=} & \exists s_m, ls_m \bullet \\ & & \quad P[s_m, ls_m / s', ls'] \wedge Q[s_m, ls_m / s, ls] \\ c * P & \widehat{=} & \mu L \bullet (P; L) \lhd c \rhd II \\ P \lhd c \rhd Q & \widehat{=} & c \wedge P \vee \neg c \wedge Q \end{array}$$

Here, the definition of $\lhd \_ \rhd$ and $\_ * \_$ are entirely standard, of course. We get the following very straightforward laws:

$$\begin{array}{rcl} II; P & = P = & P; II \\ s' = s; A & = A = & A; s' = s \\ c * P & = & (P; c * P) \lhd c \rhd II \end{array}$$

C. UTCP Semantics Overview

We view the semantics of a concurrent program as being a collection of all its atomic actions, each with an associated guard that enables it, those guards being based on the presence or absence of labels from the global label set (ls). An enabled atomic action will run when its input label (in) is in ls, at which point in time it will make changes to the global shared variable state (s), remove its in label from ls, and add its output label (out) to that set. A key point here to note is that every construct has both a input (start) label, and a output (finish) label, and its semantics is defined solely in terms of those. The output label in particular, belongs to the construct itself and is not the label (or labels) of "whatever comes after".

D. Atomic Action Semantics

We use sets in two key ways: checking for membership/subset inclusion; and updating by simultaneously adding and removing elements:

$$A \ominus (B \lhd\!\rhd C) \quad \widehat{=} \quad (A \setminus B) \cup C$$

Let us consider an atomic state change operation, viewed as a before-after relation on $State$:

$$A(s, s') \quad : \quad State \leftrightarrow State$$

We can lift this into an atomic concurrent action by adding in the appropriate behaviour w.r.t to in, out, ls and ls':

$$\mathbf{A}(A) \quad \widehat{=} \quad in \in ls \wedge A \wedge ls' = ls \ominus (\{in\} \lhd\!\rhd \{out\})$$

To keep expression size manageable and clutter-free we use a number of shorthands, viz., $ls(in)$ for $in \in ls$, $ls(a, b)$ for $\{a, b\} \subseteq ls$, ℓ for $\{\ell\}$ (when it is clear a set is expected), and $(a, b \lhd\!\rhd c, d)$ for $(\{a, b\} \lhd\!\rhd \{c, d\})$. Given this shorthand we now write the atomic action semantics as:

$$ls(in) \wedge A \wedge ls' = ls \ominus (in \lhd\!\rhd out)$$

A special case of this is the $Idle$ construct:

$$\begin{array}{rcl} Idle & \widehat{=} & \mathbf{A}(s' = s) \\ & = & ls(in) \wedge s' = s \wedge ls' = ls \ominus (in \lhd\!\rhd out) \end{array}$$

An atomic action has no need of the label generator g so simply ignores it. The situation with language composites is more complex, as we shall see.

E. Composite Action Semantics

For composite language constructs to work, we require that the context of each component is somehow "informed" of how it is being situated. We consider this the semantic responsibility of the composite itself, and not something the components need to consider.

Let us consider sequential composition ($P \mathbin{;;} Q$). In effect, this operator has to glue its components so that the out label of the first corresponds to the in label of the second. In effect it needs to generate a fresh label using g and use this to replace the out of P and the in of Q. Then we need to split the generator in two and pass those bits into P and Q for their own label generation needs. The need for such generators arises because we can't use existential quantification to hide a label, because they need their presence in, or absence from, ls to be globally visible. We need these generators to have certain properties that ensure all generated labels are unique, and it is to this that we now turn.

1) Label Generation: We consider two operations that can be applied to a generator. The first (new) returns a label, and a modified generator, for further use. The second ($split$) breaks a generator into two new generators that will produce disjoint sets of labels. To avoid long nested calls of new, $split$ and projections π_1, π_2, we define the following terse label and generator expression syntax:

$$\begin{array}{rcll} g \in GVar & & \text{Generator variables} \\ G \in GExp & ::= & g \mid G_{:} \mid G_1 \mid G_2 \\ L \in LExp & ::= & \ell_G \end{array}$$

Here $G_{:}$ denotes the generator left once new has been run on G, with ℓ_G denoting the label so generated. Expressions G_1 and G_2 denote the two outcomes of applying $split$ to G. We use $labs(G)$ to denote all the labels that G can generate and we require the following laws to hold:

$$\begin{array}{rcl} labs(G) & = & \{\ell_G\} \cup labs(G_{:}) \cup labs(G_1) \cup labs(G_2) \\ \ell_G & \notin & labs(G_{:}) \\ \emptyset & = & labs(G_1) \cap labs(G_2) \end{array}$$

The simplest model for a generator that satisfies the above constraints is one that represents the label ℓ_G by the expression G itself. The reason for this shorthand is that without it we would have to write something like the following

$$\pi_1(new(\pi_2(new(\pi_2(split(\pi_2(new(\pi_1(split(g)))))))))).$$

instead of $\ell_{g1:2:}$.

2) Sequential Composition: For sequential composition ($P \mathbin{;;} Q$) we use the generator g first to create the label (ℓ_g) that connects out of P to in of Q, and then we split the leftover generator ($g_{:}$) and give one ($g_{:1}$) to P, and the other ($g_{:2}$) to Q. We simply use substitution to replace the static parameters of both P and Q by the appropriate generator and label expressions.

$$P \mathbin{;;} Q \;\widehat{=}\; P[g_{:1}, \ell_g/g, out] \vee Q[g_{:2}, \ell_g/g, in]$$

We group the appropriately transformed components using disjunction, as per the UTPP and action systems approach.

3) Parallel Composition: Initially, parallel composition appears easy: simply split the generator and pass to each part, but leave in and out alone:

$$P[g_1/g] \vee Q[g_2/g]$$

However this does not work—consider if P is atomic and runs first: the in is removed from, and out added to ls, effectively disabling Q. Instead we need to seperate out the in and out labels of P and Q, and introduce two new atomic "actions": one to split in into two new start labels; and another to merge finish labels into out. These split and merge actions do not alter state s. We need to split the generator g into two parts (g_1, g_2) and generate two labels (start: ℓ_{g1}, ℓ_{g2}, finish: $\ell_{g1:}, \ell_{g2:}$) from each before passing them $(g_{1::}, g_{2::})$ into P and Q.

$$\begin{array}{rcl} Split(\ell_a, \ell_b) & \widehat{=} & ls(in) \wedge s' = s \wedge ls' = ls \ominus (in \Big\{\!\!\Big\} \ell_a, \ell_b) \\ Merge(\ell_a, \ell_b) & \widehat{=} & ls(\ell_a, \ell_b) \wedge s' = s \\ & & \quad \wedge ls' = ls \ominus (\ell_a, \ell_b \Big\{\!\!\Big\} out) \end{array}$$

So, the parallel composition is a disjunction between $Split(\ell_{g1}, \ell_{g2})$, the two components with appropriate relabelling, and $Merge(\ell_{g1:}, \ell_{g2:})$.

$$\begin{array}{rcl} P \parallel Q & \widehat{=} & Split(\ell_{g1}, \ell_{g2}) \\ & & \vee P[g_{1::}, \ell_{g1}, \ell_{g1:}/g, in, out] \\ & & \vee Q[g_{2::}, \ell_{g2}, \ell_{g2:}/g, in, out] \\ & & \vee Merge(\ell_{g1:}, \ell_{g2:}) \end{array}$$

4) Conditionals: For the conditional, as only one arm will run, we can share out, but we need a split on in that uses the condition c.

$$\begin{array}{rcl} Cond(\ell_a, c, \ell_b) & \widehat{=} & ls(in) \wedge s' = s \\ & & \quad \wedge ls' = ls \ominus (in \Big\{\!\!\Big\} \ell_a \lhd c \rhd \ell_b) \end{array}$$

So we split the generator g, yielding g_1 and g_2, and produce one start label from each (ℓ_{g1}, ℓ_{g2}), and then pass the two remaining generators ($g_{1:}, g_{2:}$) into P and Q as appropriate. We then have a conditional-split "action" that converts in into ℓ_{g1} or ℓ_{g2} as determined by the condition.

$$\begin{array}{rcl} P \blacktriangleleft c \blacktriangleright Q & \widehat{=} & Cond(\ell_{g_1}, c, \ell_{g2}) \\ & & \vee P[g_{1:}, \ell_{g1}/g, in] \vee Q[g_{2:}, \ell_{g2}/g, in] \end{array}$$

5) Iteration: Iteration is quite straightforward, as we view it as a conditional loop unrolling. We generate a label (ℓ_g) for the entry-point to P, pass the remaining generator ($g_{:}$) into P and identify P's exit with the loop entry.

$$\begin{array}{rcl} c \circledast P & \widehat{=} & ls(in) \wedge s' = s \wedge ls' = ls \ominus (in \Big\{\!\!\Big\} \ell_g \lhd c \rhd out) \\ & & \quad \vee P[g_{:}, \ell_g, in/g, in, out] \end{array}$$

F. Running a concurrent program

So far the semantics we have written simply views a concurrent program as a big disjunction of atomic actions that use labels in a particular way. This is a very static view of the program meaning. To get a dynamic semantics we need to embed the above static view, with appropriate initialisation, into a loop that repeatedly runs the static disjunction until a suitable (label-based) termination condition is reached. We shall denote by $run(P)$ the result of adding dynamism to a static view P in this way. We produce $run(P)$ by using the generator g to create two labels ℓ_g and $\ell_{g:}$, and then pass the remaining generator $g_{::}$ into the (static) program body P. We use ℓ_g to replace in, and $\ell_{g:}$ to replace out in P. We put this into a loop which keeps running so long as $\ell_{g:}$ is not in ls.

$$run(P) \quad \widehat{=} \quad (\neg ls(\ell_{g:}) * P[g_{::}, \ell_g, \ell_{g:}/g, in, out])[\ell_g/ls]$$

At the very top level, we initialise ls to be $\{\ell_g\}$. Note that we cannot push this into the substitutions on P, otherwise all that happens is that the first enabled action keeps running.

Space precludes us here from showing detailed calculations, but we have obtained the following results which have contributed to the validation of this semantics. Here the lefthand side shows P, while the righthand side shows a calculation of the expansion $run(P)$

$$\begin{array}{rcl}
Idle & \stackrel{run}{\to} & s' = s \wedge ls' = \ell_{g:} \\
\mathbf{A}(A) & \stackrel{run}{\to} & A \wedge ls' = \ell_{g:} \\
\mathbf{A}(A) \;;;\; \mathbf{A}(B) & \stackrel{run}{\to} & (A \;;\; B) \wedge ls' = \ell_{g:} \\
\mathbf{A}(A) \parallel \mathbf{A}(B) & \stackrel{run}{\to} & ((A \;;\; B) \vee (B \;;\; A)) \wedge ls' = \ell_{g:} \\
\mathbf{A}(A) \blacktriangleleft c \blacktriangleright \mathbf{A}(B) & \stackrel{run}{\to} & ((c \wedge A) \vee (\neg c \wedge B)) \wedge ls' = \ell_{g:}
\end{array}$$

Note that in each case we get a final result in terms of the atomic actions on s only, plus an assertion that we have terminated. For iteration we show a partial calculation of $run(c \circledast \mathbf{A}(B))$ with result:

$$\begin{array}{ll}
 & \neg c \wedge s' = s \wedge ls' = \ell_{g:} \\
\vee & c \wedge B \wedge \neg c' \wedge ls' = \ell_{g:} \\
\vee & (\; c \wedge B \wedge c' \wedge ls' = \ell_{g::} \\
 & \quad ;\mathbf{A}(B)[g_{:::}, \ell_{g::}, \ell_g, \ell_{g::}/g, in, out, ls] \\
 & \quad ;\neg ls(\ell_{g:}) * \\
 & \qquad (c \circledast \mathbf{A}(B)) \\
 & \quad [g_{:::}, \ell_{g::}, \ell_g, \ell_{g::}/g, in, out, ls]
\end{array}$$

Here we see the result of performing zero or one iteration, which also mentions only state, plus a third case that captures the second and subsequent iterations. In particular there is no mention in the final result of in or out.

The semantics of $run(c \circledast \mathbf{A}(A))$ we expect to have a terminating part and a non-terminating part:

$$\begin{array}{ll}
 & (\bigvee_{i \in \mathbb{N}} A^i) \wedge \neg c \wedge ls' = \{\ell_{g:}\} \vee A^\omega \wedge c \wedge \ell_{g:} \notin ls' \\
\textbf{where} & A^0 = s' = s \\
 & A^{n+1} = A; A^n \\
 & A^\omega = \text{infinite sequence of } A\text{s}
\end{array}$$

From the above we can see that a possible interpretation of our semantics as a Design is

$$\begin{array}{rcl}
ok & \widehat{=} & \ell_g \in ls \\
ok' & \widehat{=} & \ell_{g:} \in ls'
\end{array}$$

This works because every use of run initialises ls to ℓ_g and so establishes ok, and the termination condition for run is that $\ell_{g:}$ is has appeared in ls. In fact the use of the generators ensures that, for all constructs, that the labels in, out and $labs(g)$ never occur together in the label-set.

G. Healthiness Conditions

Our semantics has been designed to ensure that, for any program P with alphabet $s, s', ls, ls', in, out, g$, during any program run, the following four predicates are mutually exclusive, and exhaustive:

$$(\{in, out\} \cup lab(g)) \cap ls = \emptyset \quad (1)$$
$$in \in ls \quad (2)$$
$$lab(g) \cap ls \neq \emptyset \quad (3)$$
$$out \in ls \quad (4)$$

This observations comprise a key healthiness condition that ensures that all components get executed in a coherent manner, moving from a "sleeping state" (1), to being just started (2), running (3), just finished (4), and then possibly back to being asleep. We can accordingly define some program status predicates as follows:

$$\begin{array}{rcl}
started(P) & \widehat{=} & in \in ls \\
running(P) & \widehat{=} & \{in, out\} \cap ls = \emptyset \wedge \exists \ell \in ls \bullet \ell \in labs(g) \\
stopped(P) & \widehat{=} & out \in ls
\end{array}$$

Note that $running(\mathbf{A}(A))$ is always false, as an atomic action effectively occurs instantly.

H. Bringing It All Together

We link the weak semantics to the UTCP version by simply recognising that Weak PML simply views a PML description as being the parallel composition of all its basic tasks. We instantiate the generic state observation s in the UTCP theory with the r and h observations of the Weak theory, and define the semantics of a basic action A as:

$$\begin{array}{rcl}
N?rr!pr & \widehat{=} & rr \subseteq r \wedge \neg pr \subseteq r \\
 & & \wedge\, r' = r \cup rr \wedge h' = h \frown \langle N \rangle
\end{array}$$

Here we drop ok and ok', as its role is subsumed by ls and ls' in the UTCP theory, and instead have a basic action

return false when its required resources are not available, or all its provided resources have been so provided. We note that when a PML description consists solely of basic tasks and parallel composition, then the Weak, Flexible and Strict PML semantics coincide since the control flow allows any task to run anytime, so it is all just down to resource constraints.

I. Future Work

One key issue we face in developing a strict or flexible semantics is that the iteration construct has no explicit halting condition associated with it syntactically. The expectation is that an iteration is repeated until it brings about a state of affairs that enables *what comes after it*. We believe that the UTP approach described in this paper will allow us to produce a semantics that is compositional: the iteration simply repeats until it gets a stop signal sent by the enclosing sequential composition.

The instantiation of the flexible and strict semantics on top of UTCP still has to be done, as do the proofs regarding the laws of concurrent programs. The proof sketches done so far show indicate that the proofs will require demonstrating the existence of label bijections that respect control-flow, and that a benefit of constructing such bijections will be a corresponding operational semantics for UTCP.

We are also working on ideas for a variant of UTCP that shows promise for being fully compositional: *i.e.*, without having a static disjunction of actions, that then needs something like run to make it all dynamic. We also point out, in the spirit of unification, that this work has potential to go beyond just PML but also assist in the development of other theories of concurrency in UTP, such as pioneering work in [17] addressing the semantics of System-C.

We prefer the UTP denotational/algebraic approach to that of using operational semantics. The former typically requires redoing induction proofs when any change is made, whereas adding new features and merging models is much simpler with the latter [12, p277].

VI. Related Work

A. Process Semantics

Our motivation for wanting to formalise PML comes from its applicability in modelling clinical healthcare pathways [1] in particular, as well as its use to model certified medical software development processes. Clinical pathways are evidence-based care plans which describe in a structured way the essential steps needed to care for patients with a specific set of clinical problems.

Formalisms that model CPs, need to be able to 'reason' with process behaviour such as non-determinism, concurrency, parallelism and synchronisation. The affinity of the Petri nets formalism to represent such behaviour has contributed to its popularity to be used in the more 'formal spectrum' of CPs research. Petri nets have been used to model such work-flows [5], [7], [18]. Stochastic treatments are popular for looking at resource and time estimation, e.g. ICPA [3].

A key issue with any attempt to model such pathways is the need to allow flexibility in how they are actually implemented. Work by van der Aalst et al [6] proposes an approach (based on Petri nets) to allow the ability to distinguish between marginally different and completely different processes. Work looking for temporal similarity using ad hoc temporal constraint networks, has been reported in [2]. Other approaches include the use of fuzzy logic [19] or linear temporal logic expressions [9]. Of interest is the work of Grigori et al [8] that use the concept of anticipation which allows the execution of sequential tasks to overlap at the discretion of work-flow users where there are not specific data dependencies between them.

Some interesting recent work using BPMN [20] addresses the same issue that we do, namely being able to be flexible regarding how models get enacted. This paper looks at the boundary between strict imperative BPMN and declarative notations such as Declare [21]. These correspond to our Strict and Weak semantics. They introduce BPMN-D and give its semantics by translation to plain BPMN, whose semanitcs is based on Petri-Nets [22].

B. Concurrency Semantics

Key work was done on concurrent semantics in the 80s and 90s, with a strong focus on fully abstract denotational semantics. Notable work form this period includes that by Stephen Brookes [23] and Frank de Boer and colleagues [24]. Both looked at denotations based on the notion of sets of transition traces, these being sequences of pairs of before-after states. In order to get compositionality the traces of any program fragment had to have arbitrary "stuttering" and "mumbling" state-pairs added to capture the notion of outside interference. Full abstraction meant that the semantics had to identify programs like $skip$;; $skip$ with $skip$, while distinguishing between $x := 2$ and $x := 1$;; $x := x + 1$. This latter aspect required the language to be augmented with an atomic wrapper construct. A common feature of both was the very close linkage of the denotational semantics to the operational one. The work of Brookes [23] focussed on imperative languages with fair schedulers, while that of de Boer et.al [24] looked at a general framework ("failures of failures") that covered not just imperative programs but also constraint solving systems and *asynchronous* versions of process algebras.

As already stated earlier, the key inspiration and starting point for the work presented here was the UTPP paper [13]. This combined guarded commands [15] with the idea of action systems [14], interpreted in UTP as non-deterministic choice over guarded atomic actions, where disabled actions behave like the unit for that choice. This basic lattice-theoretic architecture for the UTPP semantics forms the foundation for the UTCP semantics presented here.

VII. Conclusions & Future Work

We have briefly described Process Modelling Language (PML) that is used to model business processes, with clinical pathways being an interesting specific case, with a view to

defining formal semantics for the notation that allows rigorous reasoning in a flexible manner. This need arises because clinical pathways in real medical practise are often interpreted in a loose manner, although also intended to be quite prescriptive.

We have presented a UTP theory of a "weak" interpretation of PML, as well as a UTCP theory of concurrency that acts as a baseline theory on top of which the "flexible" and "strict" interpretations may be constructed. A key contribution here is the use of label generators and the distinction between dynamic state-change observables, and static context-sensitive parameters.

One other benefit of using the UTP framework here is that it will ease the integration of application-specific resource semantic models, so that we can envisage complete formal analyses that address application area concerns, e.g. drug interactions in clinical pathways.

ACKNOWLEDGMENT

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855

REFERENCES

[1] H. Campbell, R. Hotchkiss, N. Bradshaw, and M. Porteous, "Integrated care pathways," *BMJ*, vol. 316, no. 7125, pp. 133–137, 1998.

[2] C. Combi, M. Gozzi, B. Oliboni, J. M. Juarez, and R. Marin, "Temporal similarity measures for querying clinical workflows," *Artif. Intell. Med.*, vol. 46, no. 1, pp. 37–54, May 2009. [Online]. Available: http://dx.doi.org/10.1016/j.artmed.2008.07.013

[3] X. Yang, R. Han, Y. Guo, J. Bradley, B. Cox, R. Dickinson, and R. Kitney, "Modelling and performance analysis of clinical pathways using the stochastic process algebra pepa," *BMC Bioinformatics*, vol. 13, no. Suppl 14, pp. S4–S4, 2012. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3439723/

[4] C. A. Ellis and G. J. Nutt, "Modeling and enactment of workflow systems," in *Application and Theory of Petri Nets 1993, 14th International Conference, Chicago, Illinois, USA, June 21-25, 1993, Proceedings*, ser. Lecture Notes in Computer Science, M. A. Marsan, Ed., vol. 691. Springer, 1993, pp. 1–16. [Online]. Available: http://dx.doi.org/10.1007/3-540-56863-8_36

[5] R. Eshuis and J. Dehnert, "Reactive petri nets for workflow modeling," in *Applications and Theory of Petri Nets 2003*, ser. Lecture Notes in Computer Science, W. van der Aalst and E. Best, Eds. Springer Berlin Heidelberg, 2003, vol. 2679, pp. 296–315. [Online]. Available: http://dx.doi.org/10.1007/3-540-44919-1_20

[6] W. M. P. van der Aalst, A. K. A. de Medeiros, and A. J. M. M. Weijters, "Process equivalence: Comparing two process models based on observed behavior," in *Proceedings of the 4th International Conference on Business Process Management*, ser. BPM'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 129–144. [Online]. Available: http://dx.doi.org/10.1007/11841760_10

[7] C. Ellis, K. Keddara, and G. Rozenberg, "Dynamic change within workflow systems," in *Proceedings of Conference on Organizational Computing Systems*, ser. COCS '95. New York, NY, USA: ACM, 1995, pp. 10–21. [Online]. Available: http://doi.acm.org/10.1145/224019.224021

[8] D. Grigori, F. Charoy, and C. Godart, "Anticipation to enhance flexibility of workflow execution," in *Database and Expert Systems Applications*, ser. Lecture Notes in Computer Science, H. Mayr, J. Lazansky, G. Quirchmayr, and P. Vogel, Eds. Springer Berlin Heidelberg, 2001, vol. 2113, pp. 264–273. [Online]. Available: http://dx.doi.org/10.1007/3-540-44759-8_27

[9] M. Pesic and W. M. P. van der Aalst, "A declarative approach for flexible business processes management," in *Proceedings of the 2006 International Conference on Business Process Management Workshops*, ser. BPM'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 169–180. [Online]. Available: http://dx.doi.org/10.1007/11837862_18

[10] D. C. Atkinson, D. C. Weeks, and J. Noll, "Tool support for iterative software process modeling," *Information & Software Technology*, vol. 49, no. 5, pp. 493–514, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2006.07.006

[11] P. O'Leary, J. Noll, and I. Richardson, "A resource flow approach to modelling care pathways," in *Third International Symposium on Foundations of Health Information Engineering and Systems*, Macau, 08/2013 2013. [Online]. Available: https://sharepoint.lero.ie/Publications/2013/Conference\%20Papers/2013-O\%27Leary-A\%20Resource\%20Flow\%20Approach\%20to\%20Modelling.pdf

[12] C. A. R. Hoare and J. He, *Unifying Theories of Programming*. Englewood Cliffs, NJ: Prentice-Hall International, 1998.

[13] J. Woodcock and A. P. Hughes, "Unifying theories of parallel programming," in *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002, Proceedings*, ser. Lecture Notes in Computer Science, C. George and H. Miao, Eds., vol. 2495. Springer, 2002, pp. 24–37. [Online]. Available: http://dx.doi.org/10.1007/3-540-36103-0_5

[14] R. J. R. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," in *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Quebec, Canada, 17–19 Aug. 1983, pp. 131–142.

[15] E. W. Dijkstra, *A Discipline of Programming*, ser. Series in Automatic Computation. Englewood Cliffs , NJ , USA: Prentice-Hall, 1976.

[16] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell, "The problem of programming language concurrency semantics," in *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, ser. Lecture Notes in Computer Science, J. Vitek, Ed., vol. 9032. Springer, 2015, pp. 283–307. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46669-8_12

[17] H. Zhu, J. He, S. Qin, and P. J. Brooke, "Denotational semantics and its algebraic derivation for an event-driven system-level language," *Formal Asp. Comput*, vol. 27, no. 1, pp. 133–166, 2015. [Online]. Available: http://dx.doi.org/10.1007/s00165-014-0309-8

[18] W. Aalst, "Three Good Reasons for Using a Petri-net-based Workflow Management System," in *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96)*, S. Navathe and T. Wakayama, Eds., Camebridge, Massachusetts, Nov 1996, pp. 179–201.

[19] O. Adam and O. Thomas, "A fuzzy based approach to the improvement of business processes," in *BPI05: Workshop on Business Process Intelligence*, M. Castellanos and T. Weijters, Eds., Nancy, France, 05. September 2005 2005, pp. 25–35.

[20] G. De Giacomo, M. Dumas, F. M. Maggi, and M. Montali, "Declarative process modeling in BPMN," in *Advanced Information Systems Engineering - 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings*, ser. Lecture Notes in Computer Science, J. Zdravkovic, M. Kirikova, and P. Johannesson, Eds., vol. 9097. Springer, 2015, pp. 84–100. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-19069-3_6

[21] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst, "DECLARE: full support for loosely-structured processes," in *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA*. IEEE Computer Society, 2007, pp. 287–300. [Online]. Available: http://dx.doi.org/10.1109/EDOC.2007.25

[22] R. M. Dijkman, M. Dumas, and C. Ouyang, "Semantics and analysis of business process models in BPMN," *Information & Software Technology*, vol. 50, no. 12, pp. 1281–1294, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.02.006

[23] S. D. Brookes, "Full abstraction for a shared-variable parallel language," *Inf. Comput.*, vol. 127, no. 2, pp. 145–163, 1996. [Online]. Available: http://dx.doi.org/10.1006/inco.1996.0056

[24] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten, "The failure of failures in a paradigm for asynchronous communication," in *CONCUR '91, 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 26-29, 1991, Proceedings*, ser. Lecture Notes in Computer Science, J. C. M. Baeten and J. F. Groote, Eds., vol. 527. Springer, 1991, pp. 111–126. [Online]. Available: http://dx.doi.org/10.1007/3-540-54430-5_84